such as mice and other pointing devices, as well as specialized devices such as data-acquisition units and controllers.

For example, a data acquisition unit might send periodic sensor readings to a PC. The controller chip's I/O pins could connect to analog-to-digital converters that convert sensor readings to digital signals. A host PC could use the USB link to request the latest readings periodically. Or the PC might send signals to control relays, motors, or other devices that the chip's I/O pins control.

Instead of just repeating what's in the chip's data sheet, I'll focus on what's important to know before you start working with the chip. I'll also explain anything that I found difficult or confusing to understand from the data sheet alone. When it's time to use the chip, check the data sheet for details.

Features and Limits

One compelling reason for choosing the '63743 for a project is inexpensive chips. Typical prices for the chip are a few dollars each in small quantities. And the chip contains an internal oscillator that eliminates the need to provide an external timing reference.

The chip is available in both through-hole (DIP) and surface-mount (SOIC) packages. If you have experience with assembly-language programming (or are willing to learn), the assembly-code instructions aren't too hard to master. The chip has 8 Kilobytes of program memory. With optimization, the code required to support USB communications can fit in 1 Kilobyte, leaving 7 Kilobytes for other functions.

The essential tool for developing is the Developer's Kit, which includes a development board, assembler, and debugging application. You'll probably also want the CY3649 Hi-Lo PROM Programmer with the adapter base and matrix card for the enCoRes, all available from Cypress.

The '63743 isn't suitable for every project. The chip is low speed, which means that you can't use bulk or isochronous transfers and the fastest maximum latency for interrupt transfers is 8 bytes per 10 milliseconds. Unlike some early controllers, the '63743 does support Interrupt OUT transfers. If

Figure 8-2: The chips in Cypress' enCoRe series have the essentials for USB communications and general port I/O.

you can get by with less memory or I/O, the series has chips with 6K of program memory and twelve I/O pins.

Inside the Chip

Figure 8-2 shows the chip's architecture. The CPU is an 8-bit RISC (reduced instruction set computer). It can access program memory, RAM, general-purpose I/O ports, and of course, a USB port. The USB port is actually an auto-switching port that supports both USB and the PS/2 interface for mice and other pointing devices. This feature is handy for designing devices that can plug into either port type. A variety of interrupt and reset sources can interrupt the CPU.

The frequency of the internal 6-Megahertz oscillator is accurate to within 1.5%, as required for low-speed USB. If an application requires a more precise clock source, the chip can instead use an external oscillator.

Figure 8-3 shows the pinouts of the '63743 and the '63723, which has four fewer I/O pins.

Memory

The on-chip memory of the '63743 consists of 8 kilobytes (0000h to 1FFFh) of OTP PROM for program storage and 256 bytes of RAM (00h to FFh) for temporary data storage. There are also 34 byte-wide I/O registers, each with a defined purpose.

The organization of the program memory is similar to that of other micro-controllers. Program execution begins at 00h. Addresses 00h and 01h contain a jump to the address where the main program code begins. Addresses 02h through 17h are interrupt vectors that hold the addresses to jump to when one of the chip's eleven interrupts occurs. Here is an example interrupt-vector table in firmware:

```
ORG  00h
jmp  reset        ; device reset
jmp  bus_reset    ; USB reset interrupt
jmp  error        ; 128-microsecond interrupt
jmp  1ms_timer    ; 1.024-millisecond interrupt
jmp  endpoint0    ; Endpoint 0 interrupt
```
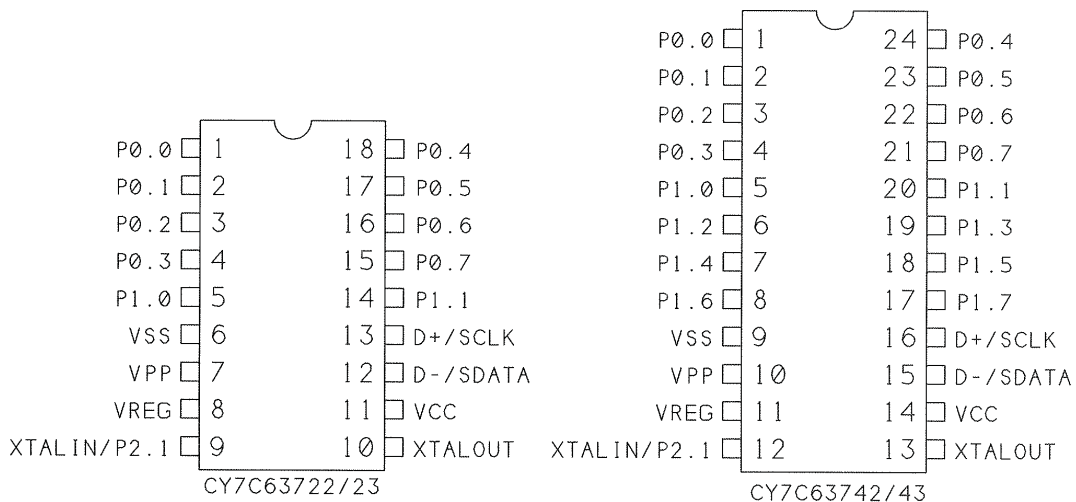

Figure 8-3: The enCoRe series includes chips with 12 and 16 I/O pins.

```
jmp   endpoint1   ; Endpoint 1 interrupt
jmp   endpoint2   ; Endpoint 2 interrupt
jmp   spi         ; SPI interrupt
jmp   capture_a   ; Capture timer A interrupt
jmp   capture_b   ; Capture timer B interrupt
jmp   gpio        ; GPIO interrupt
jmp   wakeup      ; Wake-up interrupt
```

Each interrupt vector jumps to the location specified by a label. Unused interrupts should never occur, but the firmware should include jumps even for these interrupts. A typical interrupt-service routine (ISR) for an unused interrupt would just return the firmware to the calling location with registers unchanged.

The interrupt vectors are stored in order of priority, with the highest priority at 0002h. Program memory from 0018h to 1FDFh is available for storing the rest of the code.

The 256 bytes of RAM must hold two data stacks and 8 bytes each of buffer data for Endpoints 0, 1, and 2 (if all are used), as well as any other temporary data (Figure 8-4). The endpoint buffers use addresses E8h through FFh.

The stacks are last in, first out (LIFO) structures for short-term storage of addresses and register contents. The RAM has two pointers for accessing the two stacks. The Program Stack Pointer (PSP) begins at 00h on reset and grows up, while the Data Stack Pointer (DSP) may be set by firmware to E8h or lower and grows down. The firmware needs to be sure that the stacks don't grow so large that they bump into each other in the middle. To reserve general-purpose RAM for other uses, such as storage for variables, set the DSP to an address lower than E8h. This frees the locations from that address through E7h for other uses without having to worry that one of the stacks will overwrite them.

The Program Stack Pointer

The Program Stack Pointer (PSP) holds the address the code will jump to on returning from a call to a subroutine or interrupt-service routine. For interrupts, the PSP also stores the states of the zero and carry flags. The firmware
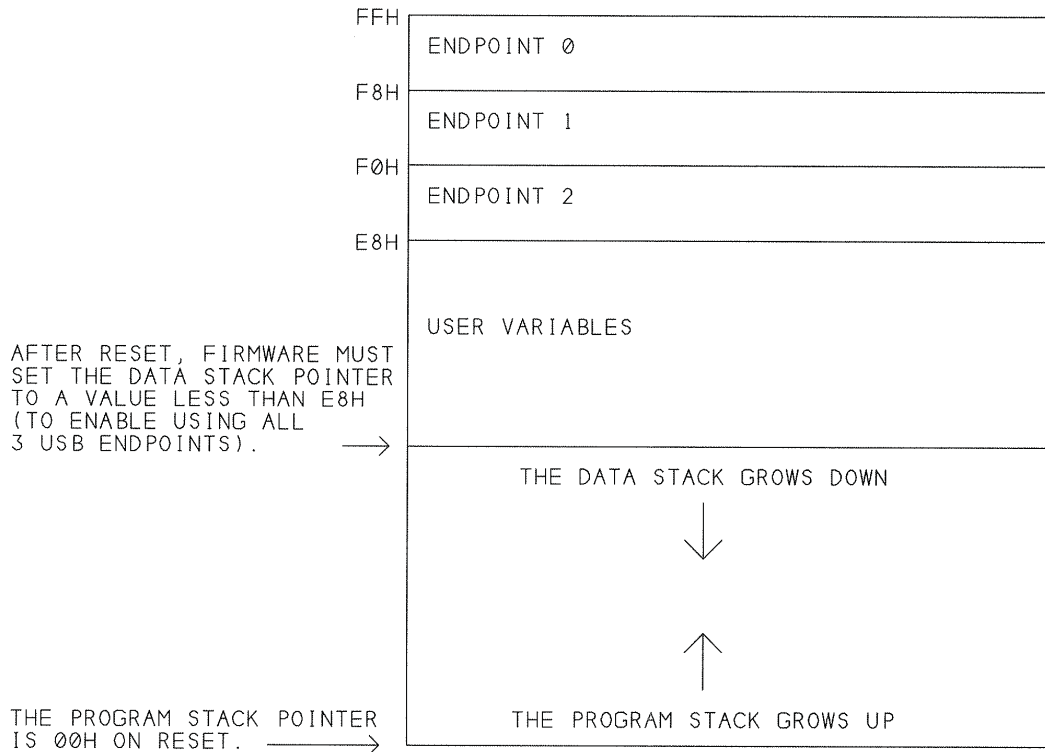
```
                          FFH  ┌──────────────────────────────────┐
                               │ ENDPOINT 0                       │
                          F8H  ├──────────────────────────────────┤
                               │ ENDPOINT 1                       │
                          F0H  ├──────────────────────────────────┤
                               │ ENDPOINT 2                       │
                          E8H  ├──────────────────────────────────┤
                               │                                  │
                               │ USER VARIABLES                   │
AFTER RESET, FIRMWARE MUST      │                                  │
SET THE DATA STACK POINTER      │                                  │
TO A VALUE LESS THAN E8H        │                                  │
(TO ENABLE USING ALL            │                                  │
3 USB ENDPOINTS).       ───→    ├──────────────────────────────────┤
                               │     THE DATA STACK GROWS DOWN    │
                               │                                  │
                               │               ↓                  │
                               │                                  │
                               │               ↑                  │
                               │                                  │
THE PROGRAM STACK POINTER       │     THE PROGRAM STACK GROWS UP   │
IS 00H ON RESET.  ───→          └──────────────────────────────────┘
```

Figure 8-4: The enCoRe's RAM contains the USB endpoint buffers, the program and data stacks, and whatever variables the firmware requires.

doesn't have to do anything to manage the PSP. It's all done automatically by the hardware and the CALL, RET, and RETI instructions.

On reset, the PSP points to 00h. The PSP can handle multiple, nested subroutines and interrupts. Each routine returns to the instruction after the last instruction that executed before the call.

For example, if the PSP is pointing to 00h when an instruction in program memory calls a subroutine, the CALL instruction will cause the PSP to save the address of the following instruction in addresses 00h and 01h. The CALL also increments the PSP by two bytes (to 02h in the example) so it's ready to store another location if needed. The RET instruction that returns from the routine places the value pointed to by the PSP in the program counter and decrements the PSP by two. Program execution then continues where it left off before the routine was called.

USB Complete

The same thing happens in interrupt-service routines, except that the values of the zero and carry flags are also saved and restored.

The Data Stack Pointer

The Data Stack Pointer (DSP) holds data stored by PUSH instructions. For example, PUSH A stores the contents of the accumulator on the data stack. The DSP decrements one byte before storing a byte. A POP instruction removes the most recently stored byte and increments the DSP.

The default value of DSP on reset is *not* where it should remain. Unless the chip isn't using USB at all, the firmware *must* set the DSP to a new value before doing any PUSH instructions. On reset, the DSP is 00h. From here, the first PUSH instruction would cause the DSP to decrement to the top of RAM (FFh), which is byte 7 in Endpoint 0's buffer. For this reason, before pushing any bytes, the firmware should set the DSP pointer to E8h or lower:

```
; Store the DSP's new beginning address
; in the accumulator.
mov A, 70h
; Swap the contents of the accumulator with the DSP.
swap A, dsp
```

Use a lower value if you want to reserve more bytes for firmware use, or a higher value the firmware needs fewer bytes.

USB Communications

The firmware monitors and controls the serial interface engine (SIE) by accessing registers. There are nine registers whose functions relate directly to USB communications: an address register, three endpoint mode registers, three endpoint counter registers, a status and control register, and an interrupt-enable register.

Device Address

The **USB Device Address Register** holds the 7-bit address assigned by the host during enumeration. The firmware must detect the Set_Address

request, send a handshake in response to the request, and store the received address in this register. Bit 7 must be set to 1 to enable the serial interface engine to respond to USB traffic.

Modes

The **USB Endpoint 0 Mode Register** contains information about the last received data packet at Endpoint 0. Both the SIE and firmware can change the register's contents.

Three PID bits indicate the type of the transaction's token packet: Setup, IN, or OUT. During the data phase of a Setup transaction, the SIE sets the Setup bit to 1. To prevent incoming data from being overwritten, the chip doesn't allow firmware to write to any USB buffer while the Setup bit is 1. Firmware can't change this bit until all of the transaction's data bytes have been received.

The ACK bit is set when a transaction completes with ACK.

Four Mode bits determine how the SIE will respond to Setup, IN, and OUT transactions. Depending on the type of transaction, the firmware can request the SIE to return ACK, NAK, Stall, a 0-byte data packet, or nothing at all. In some cases, the SIE changes the mode after a transaction's ACK. For example, when the mode is Ack OUT, after returning an ACK in response to receiving OUT data, the SIE sets the mode to Nak OUT. This gives the firmware time to retrieve the data that was ACKed. After retrieving the data, the firmware can change the mode bits back to Ack OUT to enable accepting new data at the endpoint.

For me, understanding the use of these mode bits was the most confusing part in using these chips. Cypress provides four pages of documentation about how the chip responds in every circumstance. I found it useful to group the modes according to what type of endpoint would use them, and in what situations. Table 8-3 shows the modes used by Endpoint 0. Each of these modes accepts Setup transactions, as control endpoints must.

The complements to Endpoint 0's mode register are the **USB Endpoint 1 Mode Register** and **USB Endpoint 2 Mode Register**. These have the same

Table 8-3: Modes used by Endpoint 0 in the USB Endpoint 0 Mode Register. Endpoint 0 must accept Setup transactions.

Mode	Encoding	Response to Transaction			Mode after ACK	Typical Use
		Setup	IN	OUT		
Nak In/Out	0001	accept	NAK	NAK	same	No transfer is in progress; waiting for a Setup transaction.
Status Out Only	0010	accept	Stall	check	same	Control Read transfer, status stage. Return ACK on receiving a 0-byte data packet with the correct data toggle.
Stall In/Out	0011	accept	Stall	Stall	same	No transfer is in progress; waiting for a Setup transaction.
Ignore In/Out	0100	accept	ignore	ignore	same	No transfer is in progress; waiting for a Setup transaction.
Status In Only	0110	accept	0-byte data	Stall	same	Control Write transfer, status stage. For an IN transaction, return a 0-byte data packet.
Nak Out - Status In	1010	accept	0-byte data	NAK	same	Control Write transfer, status stage. For an IN transaction, return a 0-byte data packet.
Ack Out - Nak In	1011	accept	NAK	ACK	Nak In/Out	Control Write transaction, data stage.
Nak In - Status Out	1110	accept	NAK	check	same	Control Read transfer, data or status stage. For an IN transaction, return NAK. For an OUT transaction, return ACK on receiving a 0-byte data packet with the correct data toggle.
Ack In - Status Out	1111	accept	data	check	Nak In - Status Out	Control Read transfer, data or status stage. For an IN transaction, return data. For an OUT transaction, return ACK on receiving a 0-byte data packet with the correct data toggle.

mode and ACK bits as Endpoint 0's mode register. They don't have the PID bits because these endpoints support either IN or OUT transactions only. These registers also each have a Stall bit.

Endpoints 1 and 2 use different mode settings than Endpoint 0 because they never respond to Setup packets, while Endpoint 0 must do so. Table 8-4 shows the modes used by Endpoints 1 and 2. The table also shows how firmware can use the Stall bit to cause the SIE to return Stall in Ack In and Ack Out modes.

Endpoint Status and Control

Each of the three endpoints also has a **USB Endpoint Counter Register** that contains information about the data packet that is next to transmit, is being transmitted, or has just transmitted. Each contains a four-bit count, a data-toggle bit, and a data-valid bit.

The four Byte Count bits hold the number of data bytes in a transaction. For IN transactions, the value indicates how many bytes will be sent from the endpoint's buffer in the next transaction, not including the CRC bytes. Valid values are 0 through 8. For Setup and OUT transactions, the value indicates how many data bytes were received in the last transaction, plus the two CRC bytes. Valid values are 2 through 10. Setup and OUT counts are locked until the firmware reads the register.

For Setup and OUT transactions, the Data Valid bit is 1 if the received CRC value was correct.

The Data 0/1 Toggle bit indicates the data packet's data toggle state. For IN transactions, firmware sets the value. For Setup and OUT transactions, the SIE sets the bit to match the received data-toggle state.

USB Status and Control

The **USB Status and Control register** has two bits used in USB communications, four bits that USB or PS/2 communications may use, and one bit for PS/2 communications only.

The SIE sets the USB Bus Activity bit to 1 on detecting any USB activity or in other words, a non-idle bus. The firmware can use this bit along with the 1-millisecond interrupt-service routine to decide whether the chip should

Table 8-4: Modes used by Endpoints 1 and 2 in their USB Endpoint Mode Registers. Endpoints 1 and 2 don't accept Setup transactions.

Mode	Encod-ing	Response to Transaction			Mode after ACK	Typical Use
		Setup	IN	OUT		
Disable	0000	ignore	ignore	ignore	-	The endpoint is disabled.
Nak Out	1000	ignore	ignore	NAK	-	An OUT endpoint isn't ready to receive data.
Ack Out (Stall=0)	1001	ignore	ignore	ACK	Nak Out	An OUT endpoint is ready to receive data.
Ack Out (Stall=1)		ignore	ignore	stall	-	An OUT endpoint is halted.
Nak In	1100	ignore	NAK	ignore	-	An IN endpoint has no data to send.
Ack In (Stall=0)	1101	ignore	data	ignore	Nak In	An IN endpoint has data to send.
Ack In (Stall=1)		ignore	stall	ignore	-	An IN endpoint is halted.

enter the Suspend state. If the bit remains 0 for more than three milliseconds, the chip must enter the Suspend state.

The VREG Enable bit can enable 3.3V at the chip's VREG output. This output is intended for pulling up the USB's pull-up resistor to D- on the bus. Because VREG is under firmware control, code can remove and restore the output voltage to simulate device removal and attachment. VREG's output impedance is about 200 ohms, so the resistor's value should be 1.3K to meet the 1.5K specification.

The USB Reset - PS/2 Activity Interrupt Mode bit selects whether to interrupt on a USB reset or on PS/2 activity.

Three Control bits enable firmware to set the USB or PS/2 lines to specific states, including USB's J, K, and SE0 states. If the host has previously enabled a device's Remote-Wakeup ability with a Set_Feature request, the firmware can use the Force-K state to send a Resume signal to tell the host that the device wants to communicate. Chapter 19 has more on resume signaling.

The PS/2 Pullup Enable bit can enable internal pull-up resistors on the SCLK and SDATA lines used in PS/2 communications.

The **Port 2 Data Register** holds the states of four read-only bit values at an auxiliary input port (Port 2). Two bits are the states of D+ and D- when using USB, or the states of SCLK and SDATA when using PS/2. The other two bits can sometimes serve as general-purpose inputs. If the pull-up on USB's D- uses an external voltage source or if the device doesn't support USB, the VREG output can be disabled and the pin can serve as a general-purpose input whose state is read at P2.0. When the internal clock is enabled, there is no timing reference at XTALIN, and this pin can serve as a general-purpose input whose state is read at Bit P2.1.

The final USB-related register is the **USB Endpoint Interrupt Enable Register**, which enables interrupts for Endpoints 0, 1, and 2. I cover this register in more detail below, under Interrupt Processing.

Other I/O

In addition to the USB port, the enCoRe has built-in support for three other I/O interfaces. Firmware can use the general-purpose ports for any purpose. Some of the general-purpose bits can function as an SPI synchronous serial interface. And the USB interface is switchable between USB and a PS/2 interface.

General-purpose I/O

For interfacing to circuits besides the USB port, the chip has 16 versatile I/O pins on two 8-bit ports. Each can function as an input or output. Inputs can have pull-ups or not, and CMOS or TTL thresholds. Outputs can be CMOS with selectable driver strength or open drain. Each input can trigger an interrupt. A data register and two mode registers for each port control the configuration of each pin.
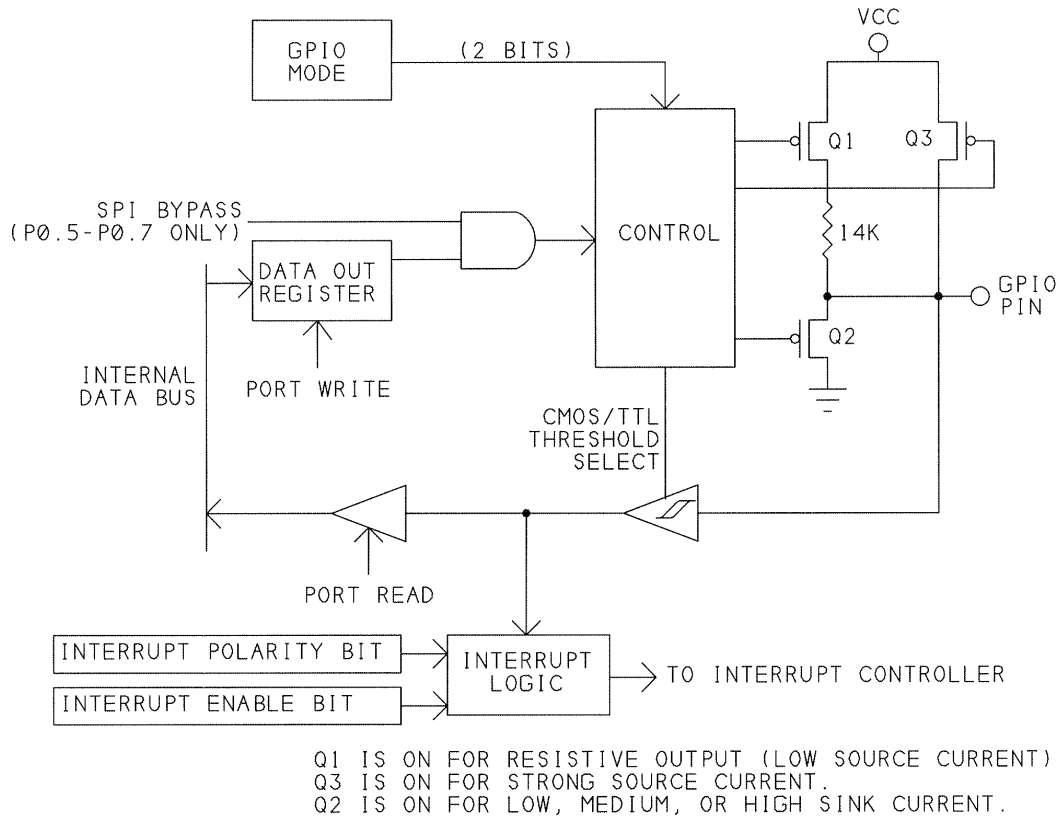
Figure 8-5: Two GPIO register bits for each pin determine whether the pin is an input or output and the amount of source and sink current an output is capable of.

The Circuits Inside

Figure 8-5 shows the circuits inside each port pin. Table 8-5 shows the effects of combinations of settings.

To configure a bit as an input, the firmware writes 0 to the matching bits in the Mode 0 and Mode 1 registers. For TTL input thresholds, write 1 to the Data bit; for CMOS, write 0. A TTL low input must be 0.8V or less, and a TTL high input must be 2.0V or greater. CMOS input thresholds are centered at around half the power-supply voltage. For low-to-high transitions, the thresholds are 40% and 60% of the supply voltage. For high-to-low transitions, the thresholds are slightly lower. This adds hysteresis to keep inputs from oscillating on noisy or slowly changing inputs.

Table 8-5: Two Mode bits and a Data bit determine the configuration and state of each general-purpose I/O bit.

Register			Output State	Output Drive Strength	Input Threshold
Data	Mode 0	Mode 1			
0	0	0	undefined	high impedance	CMOS
1	0	0	undefined	high impedance	TTL
0	0	1	0	medium (8 mA) sink current	CMOS
1	0	1	1	strong (2mA) source current	CMOS
0	1	0	0	low (2 mA) sink current (open drain on)	CMOS
1	1	0	1	resistive (14K pull-up, low source current)	CMOS
0	1	1	0	high (50 mA) sink current	CMOS
1	1	1	1	strong (2 mA) source current	CMOS

The other modes control the strength of the source and sink currents for outputs. Any output pin can sink up to 50 milliamperes, but only one pin can do so at a time. The combined sink current for all pins shouldn't exceed 70 milliamperes. For source current, the combined maximum is 30 milliamperes. Use current-limiting resistors to limit the output current.

Interrupts

A transition on a GPIO pin can cause an interrupt. Additional register bits configure the pin's interrupt capability. Writing 1 to a pin's bit in the **GPIO Interrupt Enable Register** enables a transition on the pin to trigger a GPIO interrupt. The GPIO bit in the **Global Interrupt Enable Register** must be set to 1 as well. A pin's bit in the **GPIO Interrupt Polarity Register** determines whether a rising (1) or falling (0) edge triggers the interrupt.

All of the GPIO pins share an interrupt, so the firmware may need to determine which pin caused the interrupt. It can do so by reading the port. The interrupt latency, or time it takes for the CPU to enter the interrupt-service routine, is under 3 microseconds, so an interrupt signal should be greater than 3 microseconds wide if the interrupt-service routine needs to detect which pin caused the interrupt.

SPI Port

The enCoRe includes hardware support for an SPI (Serial Peripheral Interface) port. SPI is a synchronous serial interface suitable for short-range communications, often on the same circuit board, though cables of ten feet or so shouldn't be a problem in most environments. Compared to USB, SPI doesn't require nearly as much support in hardware or code, so it's used by many simple and inexpensive chips.

Chips with SPI interfaces include serial EEPROMs and analog-to-digital converters. The enCoRe's Development System includes a couple of SPI peripherals that can connect to the chip. Motorola introduced SPI, so the 68HC11 and other Motorola microcontrollers have SPI interfaces. A peripheral that needs more processing power than the enCoRe could use an enCoRe to manage USB communications and use the SPI interface to pass information between the enCoRe and another microcontroller.

An SPI bus has one master and one or more slaves. As with USB's host, the master initiates all SPI traffic. The enCoRe's SPI can function as a master or slave. The number of wires varies with the application. In addition to a common ground, an SPI interface has MISO (master in, slave out), MOSI (master out, slave in), and SCK (serial clock) lines. When there is more than one slave connected, each must also have an *SS (slave select) line. If there is just one slave, *SS can often be tied low at the slave to select it permanently.

On a master, MOSI, SCK, and any *SS pins are outputs and MISO is an input. On a slave, MISO is an output and MOSI, SCK, and *SS are inputs.

On the enCoRe, the SPI interface uses GPIO pins. Four pins have assigned functions: MOSI is P0.5, MISO is P0.6, and SCK is P0.7. On a slave, *SS is P0.4. On a master, the *SS outputs can be any spare GPIO pins.

The hardware handles the clocking and sending and receiving of the SPI data bits. A communication consists of the master writing one or more bytes to a slave, followed by an optional reply. For example, to write a byte to serial EEPROM, the master sends a write instruction, followed by an address and data. The slave sends nothing. To read a byte from EEPROM,

the master sends a read instruction followed by an address, and the slave sends the data in reply.

Writing to the **SPI Data Register** fills a transmit buffer, which causes the data to load into a shift register for transmitting. Received SPI data is loaded into a receive buffer, where the firmware can retrieve it by reading the SPI Data Register.

The enCoRe's interface is flexible enough to communicate with just about any SPI chip. An **SPI Control Register** enables the firmware to select master or slave mode, a clock frequency from 62.5 Kbits/sec. to 2 Mbits/sec., and a clock polarity and phase. The clock polarity and phase select the clock's idle state (0 or 1) and whether data is written and read on rising or falling clock edges. Some SPI chips support only master or slave or a single clock phase and polarity.

Two additional bits in the SPI Control Register indicate when the transmit buffer is full and when an 8-bit transfer is complete. Completing a transfer also triggers an SPI interrupt so the firmware can get ready for another transfer.

The PS/2 Interface

Although this book is about USB, I shouldn't entirely neglect the enCoRe's PS/2 option. The term PS/2 can refer to the mouse, keyboard, or parallel-port interface IBM included years ago in its model PS/2 computer. In this case, we're talking about the mouse interface, which became a favored alternative to the serial (RS-232) and bus interfaces that were the options until USB came along.

A PS/2 mouse uses a synchronous serial interface that has a single data line and a clock line. The interface also has +5V and ground lines. The device provides the clock for communications in both directions. The device sends mouse data synchronized to the clock pulses. The data format uses 11 bits: a Start bit of 0, eight data bits sent least significant bit first, an odd parity bit, and a Stop bit of 1. The host reads the data on the clock's falling edge. As with a USB mouse, the data contains information about button presses and the amount and direction of mouse movement.

A long low on the data line tells the device that the host wants to send a command and generates a PS/2 interrupt in the device.

Having an interface that supports both USB and PS/2 makes it easy to design a pointing device that can use either. The device will need firmware to support both. For PS/2, the firmware is responsible for writing each clock pulse and data bit by setting Control bits in the USB Status and Control Register. Of course, a design can also use only USB, only PS/2, or even neither.

Other Chip Capabilities

The enCoRe has many other capabilities worthy of mention. Timer functions enable performing periodic tasks and measuring intervals. Many event types can trigger interrupts. And several registers enable monitoring and controlling the CPU and managing power.

Timer Functions

The chips have hardware support for a variety of timing functions, including generating interrupts for periodic tasks and measuring intervals.

Performing Periodic Tasks

For tasks to be done periodically, there are three options: the 1-millisecond, 128-microsecond, and Wake-up timer interrupts. The Wake-up interrupt provides less precise, but longer, timing intervals than the other two timers. If the chip is in the Suspend state, this interrupt will wake it. But firmware can also use this interrupt to perform periodic tasks when the chip isn't suspended.

The timing interval of the Wake-up interrupt is the chip's tWAKE period multiplied by the value indicated by three Wake-up Timer Adjust bits in the **Clock Configuration Register**. The available values are the eight powers of 2 from 1 through 128. The tWAKE value varies with the supply voltage and temperature, and can range from 1 to 5 milliseconds. So for example, if tWAKE is 128, the interval may be anywhere from 128 to 640 milliseconds.

To select an interval more precisely, the firmware can enable the Wake-up timer, use the chip's free-running timer to measure the interval, and select the Wake-up Timer Adjust value that most closely matches the desired interval.

With any of these timers, to time a longer interval, the firmware can maintain a counter in the interrupt-service routine. The routine increments the counter on each interrupt until the desired number of intervals has elapsed.

Measuring Intervals

The enCoRe has a free-running timer that provides a way to measure intervals and timer capture registers that enable measuring the time between events at I/O pins.

The 12-bit free-running timer increments once per microsecond, The timer rolls over on a count of FFFh, enabling firmware to measure periods up to 4.096 milliseconds (or longer by cascading counts). The count is stored in two registers. The firmware can read just one register at a time, yet it will want to know the states of all 12 bits at the same time. To make this possible, reading the **Timer LSB (least significant byte) Register** also loads the timer's upper four bits into a temporary register. Reading the **Timer MSB (most significant byte) Register** reads the temporary register. So sequential reads of these two registers gives the count at the time of the first read.

The chip can also measure intervals between events at the GPIO pins Port 0.0 (Capture A) and Port 0.1 (Capture B). Six registers configure the timers and hold the results, which can correspond to the times of rising and falling edges at each pin.

The **Capture Timers Configuration Register** has three functions. Four bits enable interrupts on the rising and falling edges of Capture A and B. One bit selects whether to save the time of the first edge or the most recent edge. Three bits select a prescale value that determines which 8 of the free-running timer's 12 bits are saved on an interrupt. Using lower bits gives better precision but shorter range, while higher bits give longer range but less precision.

The **Capture Timers Status Register** indicates whether a rising or falling edge has occurred on Capture A or B. The four **Capture Timer Data Registers** hold the timer counts for rising and falling edges at the two port pins. The difference between the counts stored at two events equals the time in microseconds between them.

Interrupt Processing

The firmware uses two registers to control which interrupts are enabled, plus two additional registers to enable individual GPIO interrupts. The **USB Endpoint Interrupt Enable Register** has three bits that enable interrupts for Endpoints 0, 1, and 2. The **Global Interrupt Enable Register** enables the other interrupt sources: Wake up, General-purpose I/O, Capture Timer A, Capture Timer B, SPI, 1.024-millisecond timer, 128-microsecond timer, and USB Reset or PS/2 Activity. Writing 1 to an interrupt's bit enables the interrupt, while writing 0 masks, or disables, the interrupt.

Interrupt Service Routines

When an interrupt occurs, the chip's hardware disables all interrupts, clears the Global Interrupt Enable bit and jumps to the interrupt's assigned interrupt-vector location in program memory. This location typically contains a jump to an interrupt-service routine. The interrupt-service routine is responsible for carrying out whatever needs to be done in response to the interrupt's event and for ensuring that all registers are in the expected states on exiting the routine.

On entering an interrupt-service routine, the hardware automatically stores the Program Counter's value and the states of the Carry and Zero flags. On exiting the routine, these values are automatically restored. So the interrupt-service routine can do what it wants with these values, and other code won't be affected. The firmware is responsible for saving and restoring any other values that need to be preserved. A typical example saves and restores the contents of the accumulator (A) and index register (X). Here is an example interrupt-service routine that uses push and pop to preserve the contents of these registers while also allowing the interrupt-service routine to use the registers:

```
DoNothing_ISR:
;Save the contents of the accumulator
push A
;Push the contents of the index register
push X
;Add code to service the interrupt here
;Pop values that were preserved
;in the reverse order they were saved (last first)
pop X
pop A
reti
```

GPIO Interrupts

For the general-purpose I/O (GPIO) interrupts, a **Port Interrupt Enable Register** for each port allows the firmware to enable or disable the interrupt for each I/O pin. A transition on a port pin will result in an interrupt only if several things are true:

• The GPIO bit in the Global Interrupt Enable register is set to 1.

• The pin's bit in its port's Port Interrupt Enable register is 1.

• The polarity of the transition on the port pin matches the polarity set in the pin's bit in the corresponding Port Interrupt Polarity Register.

• If any previous GPIO interrupt has occurred, that pin's state must have returned to the inactive, or non-trigger state, or the pin's bit in the Port Interrupt Enable register must have been set to 0 (and may optionally then be set back to 1). For a low-to-high interrupt trigger, the non-trigger state is low; for a high-to-low trigger, the non-trigger state is high.

USB Endpoint Interrupts

The USB endpoint interrupts trigger on sending or receiving the last packet in a transaction. In a Setup transaction, an interrupt occurs when the device returns ACK or receives a flawed data packet. In an IN transaction, an interrupt occurs on receiving the host's ACK or if the device returns a NAK or Stall. In an OUT transaction, an interrupt occurs when the device returns ACK, NAK, or Stall or receives a flawed data packet.

Timer Interrupts

The timer interrupts occur at intervals of 1.024 milliseconds and 128 microseconds. The firmware can use these interrupts for any purpose. One use for the 1-millisecond interrupt is to measure the amount of time with no USB activity to determine whether or not to enter the Suspend state.

Deciding whether to enter the Suspend state requires firmware support. The code must maintain a count of the number of milliseconds that the bus has been idle and cause the chip to enter the Suspend state when the count equals or exceeds 3. The count can be stored in any spare location in RAM.

To find out if the bus has been idle, the firmware reads the bus-activity bit in the USB Status and Control register. If the bit is 0, there has been no bus activity and the firmware should increment the suspend counter. If the bit is 1, there has been activity, and the firmware should clear the suspend counter and the bus activity bit by writing 0 to each:

```
1ms_timer:
; Sample 1-millisecond timer routine
; that checks bus activity and enters the Suspend
; state if there has been no bus activity for over
; 3 milliseconds.

push A

1ms_suspend_timer:
; To check for bus activity,
; read the bus-activity bit
; in the USB Status register.
iord usb_status
and  A, BUS_ACTIVITY
;If it's not 0, there has been bus activity.
jnz  bus_activity

;If it's 0, there has been no bus activity
;since the last 1-millisecond interrupt.
;Increment the suspend counter to keep track of
;the amount of time with no bus activity.
inc  [suspend_count]
mov  A, [suspend_count]
;Has it been over 3 milliseconds?
```

```
cmp  A, 04h
;If yes, enter the Suspend state.
jz   usb_suspend
;If no, we're finished checking for bus activity.
jmp  ms_timer_done

usb_suspend:
; Before entering the Suspend state,
; enable the Reset interrupt.
mov  A, (USB_RESET_INT)
iowr global_int

; Set the Suspend bit in the control register
; and re-enable interrupts.
iord control
or   A, SUSPEND
ei
iowr control

;On exiting Suspend, program execution begins here.
nop

; Look for bus activity.
; If there has been none, return to the Suspend state.
iord usb_status
and  A, BUS_ACTIVITY
jz   usb_suspend

; Exit the Suspend state.
; Enable the 1-milliscond and Reset interrupts.
mov  A, (1MS_INT | USB_RESET_INT)
iowr global_int

bus_activity:
; Bus activity was detected.

; Reset the Suspend counter to 0.
mov  A, 00h;
mov  [suspend_count], A

; Clear the bus-activity bit.
iord usb_status
and  A, ~BUS_ACTIVITY
```

```
iowr usb_status

ms_timer_done:
;Exit the 1-millisecond timer ISR.
pop A
reti
```

The Wake-up interrupt occurs at intervals set by firmware. If the chip is in the Suspend state, the Wake-up interrupt will wake it. The Wake-up interrupt is enabled whenever the Wake-up Interrupt Enable bit in the Global Interrupt Enable Register is 1, even if hardware or firmware has disabled interrupts.

Interrupt Status

The **Processor Status and Control Register** has two bits that relate to interrupts.

The Interrupt Enable Sense bit shows whether interrupts are enabled (1) or disabled (0). Firmware can control its state with the instructions DI (disable interrupts), EI (enable interrupts), and RETI (return from interrupt-service routine and re-enable interrupts. The hardware disables interrupts on entering an interrupt-service routine and re-enables them on exiting.

When interrupts are disabled, the IRQ Pending bit in the Processor Status and Control register indicates when an interrupt has occurred but has been ignored because interrupts are disabled. The bit remains set until the interrupt(s) are enabled and serviced.

CPU Status, Control, and Clocking

The **Processor Status and Control Register** contains seven bits that relate to the chip's overall operation. Two bits can stop the CPU, two bits relate to resets, and three bits relate to interrupts. In addition the **Clock Configuration Register** has bits that relate to resets and CPU clocking.

Halting the CPU

To stop the CPU, the HALT instruction sets the Run bit in the Processor Status and Control Register to 0. The CPU stops executing instructions until a reset occurs. The CPU resumes at address 0.

Writing 1 to the Suspend bit in the Processor Status and Control Register puts the chip in the Suspend state. The chip stops executing instructions until there is USB activity or a pending, enabled interrupt occurs. The CPU resumes at the instruction following the instruction that set the Suspend bit.

Resets

The CPU supports three types of reset: Low Voltage, Brown Out, and Watch Dog. Each is triggered by a different event. A fourth type of reset is the bus reset that a USB host may request to restart USB communications.

On a Low-Voltage or Brown-Out reset, the chip is placed in a known state: the PSP and DSP are set to 0, the USB address is set to 0, interrupts are disabled, and registers return to their default states. The GPIO, USB, and VREG pins are high impedance. USB communications are disabled. A chip using an external clock switches to the internal clock. After a short delay, program execution begins at 0. After reset, the firmware is responsible for writing the desired default values to registers and variables. After enabling USB communications, the chip has to wait to be enumerated by the host before it can do other USB communications.

A useful feature is the ability to shut the chip down automatically if the supply voltage is low and start it up again when voltage is restored. The Low-Voltage and Brown-Out resets perform this function.

A Low-Voltage Reset occurs when the supply voltage is below the low-voltage-reset voltage of 3.5 to 4.0V. This reset also acts as a power-on reset that occurs when power is first applied to the chip. The internal oscillator runs, but the chip is held in reset until the supply voltage reaches the reset threshold and 24 to 60 milliseconds has elapsed. The delay gives the supply voltage time to stabilize.

After power up, a Low-Voltage Reset occurs any time the supply voltage falls below the threshold, unless firmware has set the Low Voltage Reset Disable

bit in the Clock Configuration Register, or unless the device is in the Suspend state.

When the Low-Voltage Reset isn't enabled, the Brown-Out Reset takes over. This reset does nothing until the supply voltage is below about 2.5V. The Brown-Out Reset is also active when the chip is in the Suspend state. This enables a suspended chip to have a lower supply voltage and still preserve the states of registers and memory. If the voltage falls below 2.5V and a Brown-Out reset occurs, the chip remains in reset until the supply reaches the low-voltage reset threshold.

The Watch-Dog Reset prevents the firmware from hanging by requiring the firmware to reset a watch-dog timer periodically. If the timer isn't reset, something has gone wrong and the firmware restarts. To prevent a Watch-Dog Reset, firmware must write any value to the **Watch Dog Restart Register** at least once every 10 milliseconds. If it fails to do so, the watch-dog timer overflows and triggers a reset. This reset behaves like the Low-Voltage and Brown-Out resets, except that the chip will continue to use an enabled external clock and the reset delay is just 2 to 4 milliseconds.

The interrupt-service routine for the 1-millisecond timer might seem a natural place to write to the Watch Dog Restart Register, but it's possible for firmware to stall or get stuck in a loop while still being able to service this interrupt. So it's best to reset the watch dog in the firmware's main task loop and also in any other routines that may take longer than 10 milliseconds.

Firmware can't disable the Watch Dog interrupt. The Processor Status and Control Register has a bit that indicates if a Watch Dog reset has occurred, and a bit that indicates if a Low Voltage or Brown-out reset has occurred.

A USB Bus reset occurs when the host sends a reset by bringing both USB signal lines low for at least 10 milliseconds. This doesn't reset the CPU. It just calls the USB Bus Reset interrupt-service routine. The bus-reset routine must cause the chip to stop USB communications and wait to be enumerated. And if this is necessary, the firmware is likely to want to start fresh from 00h as it does on the other resets. Here is example bus-reset code that does this:

```
bus_reset:
```

```
;Disable USB communications, then reset the firmware.
; Return Stall to IN and OUT token packets.
mov  a, STALL_IN_OUT
iowr ep0_mode

; Enable USB address 0.
mov  a, ADDRESS_ENABLE
iowr usb_address
; Disable Endpoints 1 and 2.
mov  a, DISABLE
iowr ep1_mode
iowr ep2_mode

; Set the program stack pointer to 0.
mov  A, 00h
mov  psp, a
; Execute reset code.
jmp  reset
```

Selecting and Controlling the Clock

A very convenient feature of the enCoRe is its on-chip oscillator. There's no need to connect an external crystal or resonator unless the device needs a more precise frequency for other functions. An external clock can be a crystal oscillator or ceramic resonator, plus any required capacitors at the XTALIN and XTALOUT pins.

The Clock Configuration Register has four bits that relate to clocking the CPU. The chip always uses the internal clock on power up and on returning from a Low-Voltage or Brown-Out reset. Firmware can then set the External Oscillator Enable bit to 1 to switch the CPU to an external clock. If this bit is 0, the XTALIN pin is a general-purpose input (P2.1).

When using the internal clock, the Internal Clock Output Disable bit determines whether XTALOUT is a logic high or a 6-Megahertz clock.

When using an external clock, the External Clock Resume Delay bit selects one of two delay times when switching to the external clock or waking from the Suspend state with the external clock enabled. As a rule, ceramic resonators can use the 128-microsecond delay, while crystals will need the 4 millisecond delay.

When firmware has set the Precision USB Clocking Enable bit to 1, the clock frequency meets USB's 1.5% tolerance requirements.

Power Management

The chip requires a power supply of 4.0 to 5.5V DC.

To save power and to comply with the USB specification, the chip can enter a Suspend state that powers down everything except what's needed to detect USB activity and whatever external interrupts are enabled. The on-chip oscillator stops, so there is no clock to cause program instructions to execute. The chip just waits for an event that will end the Suspend state.

The events that will end the Suspend state are non-idle activity at the USB receiver, the triggering of an enabled interrupt at an I/O pin, an SPI slave interrupt, or a Wake-Up interrupt.

The chip enters the Suspend state by writing 1 to the Suspend bit in the Processor Status and Control Register. Program execution stops. When an event brings the chip out of the Suspend state, program execution begins at the instruction following the `iowr` instruction that suspended the chip.

The firmware can put the chip into the Suspend state at any time, but it must do so if there has been no USB activity (including low-speed keep-alive signals) for three milliseconds. And as Chapter 19 explains, a device suspended for this reason must consume very little bus power, as little as 500 microamperes in some cases.

There are some things the firmware can do to ensure the lowest possible power consumption. The firmware should set unused bits on ports 0 and 1 to pull-up mode. On 18-lead packages, this includes P1.2 though P1.7, which are not brought out to external pins. The GPIO interrupt bits in the Port 0 and 1 Interrupt Enable Registers should all be 0, even if the GPIO bit in the Global Interrupt Enable Register is 0.

USB Complete

9

Writing Firmware: the Cypress enCoRe

Whatever controller chip you select for a project, it won't be much use until you write the code that enables it to communicate with the host and the other circuits in your peripheral. In this chapter, I again use the Cypress enCoRe series an example, this time to show what's involved in writing and debugging USB firmware, including a review of development tools. Even if you're using a different chip, this chapter will give you an idea of what the process involves.

Hardware and Firmware Responsibilities

In a USB transfer, the CY7C63743's serial interface engine handles many of the tasks, but the firmware still has plenty to do. Here is a look at the responsibilities of each.

What the Hardware Does

These are the tasks the hardware does on its own:

- Detects new incoming packets.
- Translates received information from the encoded format used on the USB's data lines.
- Determines whether a transaction is directed to the chip's USB address and if not, ignores the transaction.
- For transactions with Endpoint 0, determines the transaction type (Setup, IN, or OUT) and sets a bit in the endpoint's USB Mode register to indicate which type it is.

For received data, the hardware also does the following:

- Stores valid received data in the endpoint's buffer or toggles a register bit to indicate an error in received data.
- Sets the count in the Endpoint Counter Register to match the number of received bytes.
- Stores the data-toggle state of valid received data.
- Calculates CRC values, compares them to the received CRC values, and takes action on detecting an error.
- Sends the appropriate handshake to the host.
- Triggers an interrupt so the firmware can prepare for the next transaction.

For data to be transmitted, the hardware also does the following:

- Translates data to be transmitted from the bytes in the USB buffer to the format used on the USB's data lines.
- Sends the number of bytes specified in the Endpoint Counter Register onto the USB lines in response to the host's IN token packet.
- Calculates and sends CRC bits with the data.
- Sends a data-toggle code with the data.
- On receiving a handshake from the host, triggers an interrupt.

What the Firmware Does

The firmware's job in USB communications is to supplement the hardware's capabilities and ensure that the device exchanges data as needed in both directions. The following code is adapted from Cypress' example firmware.

Endpoint 0 Interrupts

An interrupt at Endpoint 0 indicates activity that the firmware should check into. On receiving an Endpoint 0 interrupt, the firmware pushes the accumulator and index registers. The firmware checks the ACK bit in the Endpoint 0 Mode Register and exits if the transaction didn't complete with an ACK. Otherwise, the firmware checks the same register to find out whether a Setup, IN, or OUT token packet was received, then jumps to a routine to handle it:

```
endpoint0:
push X
push A

; Read the ep0_mode register to enable writing to
; the endpoint's buffer.
iord ep0_mode
; If EP0_ACK isn't set, the transaction didn't
; complete with an Ack, so exit the routine.
and  A, EP0_ACK
jz   ep0_done

; Bit 5, 6, or 7 in ep0_mode is set to indicate
; whether the transaction type is Setup, In, or Out.
; Find out which it is and jump to handle it.
iord ep0_mode
asl  A
jc   ep0_setup_received
asl  A
jc   ep0_in_received
asl  A
jc   ep0_out_received

ep0_done:
popA
popX
```

```
     reti
```

If it's a Setup transaction, the firmware determines which request it is and jumps to a routine to handle it:

```
ep0_setup_received:

; Clear the Setup bit to enable
; writing to Endpoint 0's buffer.
mov  A, NAK_IN_OUT
iowr ep0_mode

; Extract the 5-bit bmRequestType in
; Endpoint 0's byte 0.
mov  A, [bmRequestType]
; Bits 2, 3, and 4 are unused here, so set to 0.
and  A, E3h
push A
; Shift right 3 places to move bits 5, 6, 7
; into bits 2, 3, and 4's places.
asr  A
asr  A
asr  A
; Save the result.
mov  [int_temp], A
; OR the result  with the original value
; to restore bits 0, 1.
pop  A
or   A, [int_temp]
; Clear bits 5, 6, & 7 (unused).
and  A, 1Fh
; Shift left to multiply by two because the
; the index table's jumps are two bytes each.
asl  A
; Use a jump table to get the address to jump to
; to handle the request indicated in bmRequestType.
jacc bmRequestType_jumptable
```

Sending Data to the Host

When a request requires Endpoint 0 to send data to the host in the Data stage, the firmware stores two values and calls an initialize_control_read routine to get ready for the expected IN

transaction(s). The value `maximum_data_count` is the amount of data available to send.

```
initialize_control_read:
; ep0_transtype indicates the transaction type.
; The firmware uses this value to decide how to
; respond to token packets.
;
; If the firmware has jumped here,
; it's a control Read transaction:
mov  A, TRANS_CONTROL_READ
mov  [ep0_transtype], A

; Set the data toggle to 1
mov  A, DATA_TOGGLE
mov  [ep0_data_toggle], A

; Find the lesser of the requested data (in wLengthhi
; and wLengthlo) and the maximum data available
; (in maximum_data_count).
; Store this value in maximum_data_count.

; If wLengthhi > 0,
; maximum_data_count is the smaller value.
mov  A, [wLengthhi]
cmp  A, 00h
jnz  initialize_control_read_done

; If wLengthhi = 0 and wLengthlo > maximum_data_count
; maximum_data_count is the smaller value.
mov  A, [wLengthlo]
cmp  A, [maximum_data_count]
jnc  initialize_control_read_done

; Otherwise, wLengthlo is the smaller value.
mov  A, [wLengthlo]
mov  [maximum_data_count], A

initialize_control_read_done:
jmp  control_read_data_stage
```

The firmware then loads data into Endpoint 0's buffer and configures the endpoint to return the data when the host sends an IN token packet.

```
control_read_data_stage:
; Load Endpoint 0's buffer with data to send.
; Initialize the index register.
mov  X, 00h
; If all of the data has been sent, we're done.
mov  A, [maximum_data_count]
cmp  A, 00h
jz   dmabuffer_load_done

dmabuffer_load:
; Load a byte number into the buffer.
mov  A, X
; If the buffer is full, we're done.
cmp  A, 08h
jz  dmabuffer_load_done
; The data to send begins at
; (data_start + control_read_table).
mov  A, [data_start]
index control_read_table

; Use the X register to step through
; Endpoint 0's buffer.
mov  [X + ep0_dmabuff0], A
inc  X
; data_start points to the byte to send.
inc  [data_start]
; maximum_data_count is the number of bytes
; remaining to send.
dec  [maximum_data_count]
; If no bytes remain, we're done.
jz   dmabuffer_load_done
; Otherwise, loop to load more data.
jmp  dmabuffer_load

dmabuffer_load_done:
; Unlock the counter register.
iord ep0_count
; Place the number of bytes loaded and
; the data toggle value in the counter register.
mov  A, X
or   A, [ep0_data_toggle]
iowr ep0_count
```

```
; Configure Endpoint 0 to return data on the next IN
; token packet or to check for a 0-byte data packet
; in an OUT transaction.
mov  A, ACK_IN_STATUS_OUT
iowr ep0_mode

; Toggle the data toggle.
mov  A, DATA_TOGGLE
xor  [ep0_data_toggle], A

pop  A
pop  X
reti
```

If there are more data packets, the device loads these into the endpoint buffer in the same way. When the host is finished requesting data, it sends a 0-byte data packet in the Status stage. The device's endpoint responds with ACK and the firmware jumps to routine that sets the endpoint's mode and the transaction type:

```
control_read_status_stage:
; Configure Endpoint 0 to return a 0-byte data packet
; in case there is another IN packet.
mov  A, STATUS_IN_ONLY
iowr ep0_mode

; No transaction is in progress.
mov  A, TRANS_NONE
mov  [ep0_transtype], A

pop  A
pop  X
reti
```

Receiving Data from the Host

When a request requires the host to send data to Endpoint 0 in the Data stage, the firmware calls an `initialize_control_write` routine to prepare to receive data in the expected OUT transaction(s). The variables `wLengthlo` and `wLengthhi` hold the amount of data the host says it will send.

```
initialize_control_write:
```

```
; ep0_transtype indicates the transaction type.
; The firmware uses this value to decide how to
; respond to token packets.

; If the firmware has jumped here,
; the transaction type is control Write:
mov  A, TRANS_CONTROL_WRITE
mov  [ep0_transtype], A

; Initialize the data toggle to 1.
mov  A, DATA_TOGGLE
mov  [ep0_data_toggle], A

;Send ACK in response to OUT packets,
;which will contain the Control Write data.
;Send NAK in response to IN packets (not expected).
mov  A, ACK_OUT_NAK_INe
iowr ep0_mode

; Return from Endpoint 0's ISR.
pop  A
pop  X
ret  i
```

When the host sends data in an OUT transaction, the device stores the data in the endpoint's buffer and triggers an interrupt to handle it. The firmware uses the token packet and ep0_transtype value to jump to the appropriate routine:

```
control_write_data_stage:
; If the data-valid bit isn't set,
; we're done with the data stage.
iord ep0_count
and  A, DATA_VALID
jz   control_write_data_stage_done

; Compare the received data toggle
; with the expected value.
iord ep0_count
and  A, DATA_TOGGLE
xor  A, [ep0_data_toggle]
; If it's incorrect,
; we're done with the data stage.
```

```
      jnz   control_write_data_stage_done

      ; Copy the received bytes to data memory.
      ; This example copies two bytes.
      mov   A, [ep0_dmabuff0]
      mov   [data_byte_0], A
      mov   A, [ep0_dmabuff1]
      mov   [data_byte_1], A

      ;Toggle the data-toggle bit.
      mov   A, DATA_TOGGLE
      xor   [ep0_data_toggle], A

      ; If all of the data has been received,
      ; configure Endpoint 0 to send a 0-byte data packet
      ; in response to an IN packet (the transfer's status
      ; stage) or to Stall an Out packet (not expected).

      mov   A, STATUS_IN_ONLY
      iowr ep0_mode

control_write_data_stage_done:
      ; Return from Endpoint 0's ISR.
      popA
      popX
      reti
```

After the endpoint has responded to the 0-byte IN transaction in the Status stage, an interrupt triggers and the firmware re-configures the endpoint and sets ep0_transtype:

```
control_write_status_stage:
      ; Jump here if the device has received an IN token
      ; packet with ep0_transtype = TRANS_CONTROL_WRITE.
      ; The device has sent a 0-byte IN data packet to
      ; complete the transfer because ep0_mode was set to
      ; Status_In_Only at the end of the data stage.

      ; Configure Endpoint 0 to return ACK on receiving
      ; a 0-byte data packet and to return Stall on INs.
      mov   A, STATUS_OUT_ONLY
      iowr ep0_mode
```

```
; No transfer is in progress.
mov   A, TRANS_NONE
mov   [ep0_transtype], A

; Return from Endpoint 0's ISR.
pop   A
pop   X
reti
```

Handling Interrupt Transfers

The code for handling interrupt transfers at Endpoints 1 and 2 isn't as complicated, because these transfers don't have multiple stages to manage. On an IN endpoint, the interrupt triggers after the endpoint has sent data or a NAK in a transaction. Here is code that enables Endpoint 1 to respond to IN interrupts:

```
endpoint1:
push A
; Get ready for the next transaction.
; Toggle the data toggle.
mov   A, 80h
xor   [ep1_data_toggle], A

; Set the event_machine variable to indicate that
; no transaction is in progress.
mov   A, NO_EVENT_PENDING
mov   [event_machine], A

; If the endpoint has been set to Stall,
; set the mode to Stall INs and OUTs.
mov   A, [ep1_stall]
cmp   A, FFh
jnz   endpoint1_done
mov   A, STALL_IN_OUT
iowr  ep1_mode

endpoint1_done:
pop   A
reti
```

In a similar way, the interrupt-service routine for an OUT endpoint retrieves the received data (as in a Control Write transaction) and gets ready for the next transaction.

Other Responsibilities

The examples above show the essence of USB communications with the CY7C63743. There are other details, of course. For example, during control transfers the firmware must check periodically to find out if another Setup token has arrived, and if so, abandon the current transfer and start the new one. The firmware must also remember to clear the watch-dog timer in any loop that might otherwise allow the timer to run without a reset for 10 milliseconds. I also haven't covered the specifics of how to respond to each control request. Again, Cypress provides example code for the essential functions and my website (*www.Lvr.com*) has firmware examples that build on Cypress' examples.

Hardware Development Tools

For project developing for the enCoRe, Cypress offers a Development Kit for debugging code and third-party PROM programmers for storing code in the chips' PROMs.

The Development Kit

The CY3654 Development Kit enables you to test your code and circuits and find problems quickly.

The system includes a set of circuit boards (Figure 9-1) and a debugging program that together enable you to load your assembled or compiled code from a PC to the board's RAM. The RAM emulates the controller's PROM. You can run and debug code while using your PC to monitor and control program execution. Downloading to RAM makes it easy to modify the code. Manufacturers of other USB chips have similar development systems for their chips.

To use the Development Kit, you need a PC running Windows 98 or later with available USB and RS-232 ports.

The Platform Board

The Development Kit's main Platform board doesn't contain an enCoRe chip. Instead it has circuits that emulate the functions of the chip while allowing you to monitor and control program execution.

Figure 9-1 shows a typical setup. The Platform board contains the circuits that emulate the microcontroller. It has connectors for a Personality Board and an RS-232 connection to a PC. The Platform board also has a USB connector for possible future use as an alternative to the RS-232 connection.
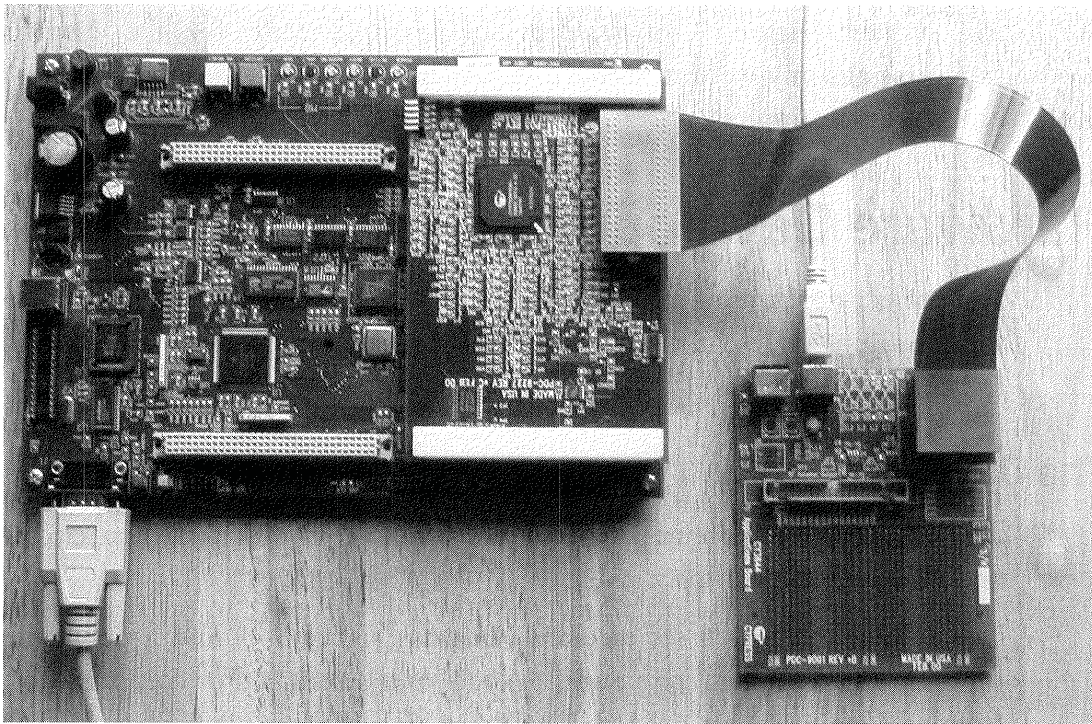
Figure 9-1: In the CY3654 Development System, a Personality Board attaches on top of the main Development Board. An RS-232 port enables communicating with the monitor program. A cable and Target Adapter connect the Personality Board to an Application Board (right), which has a USB port.

The Personality Board configures the emulator for a specific chip. A series of similar chips may share the same Personality Board. For example, all of the enCoRes use the P05 board, while the CY7C634/5/6xx chips use the P02 board.

A cable assembly connects the Personality Board to a Target Probe Adapter that in turn connects to the Application Board.

The Application Board contains the USB connector and a prototyping area. The board supports several example applications, with components for some installed. You can use your own application board in place of the one provided.

The development kit connects to a PC via both USB and RS-232 interfaces. These may, but don't have to, connect to the same PC. The USB interface of course carries the USB communications between a PC and the device's USB port. The debugger uses the RS-232 interface to send object code and to send and receive debugging information such as breakpoints and register contents. The board uses an external power supply, which is included.

The Application Board has several features for experimenting:

- Solder pads for the GPIO pins.
- A header for a cable to a logic analyzer or other circuits that connect to the GPIO pins.
- A temperature converter that uses an SPI interface (Dallas Semiconductor DS1722).
- An EEPROM that uses an SPI interface (Xicor X25020).
- Solder pads for four surface-mount LEDs, with two installed.
- Solder pads for three surface-mount push-button switches, with one installed.
- Solder pads for adding Linx Technologies' TXM and RXM RF interface modules.
- Prototyping area.

Setting Up the Development Board

Setting up the Development Board for use requires attaching five components in series. There are a few places where you can plug something in wrong, so I'll go over the steps:

1. Plug the Personality Board into the Development Board. The Personality Board rests on top of the Development Board. The bottom of the Personality Board has two headers that plug into connectors on the Development Board. The connectors are keyed so you can't plug them in backwards.

2. Plug one end of the cable assembly into the Personality Board. One end of the cable assembly has a circuit board with two 40-pin sockets (J1 and J2). These mate with the two 40-pin headers on the Personality Board. These connectors are *not* keyed, so be sure to plug the cable in correctly. The sockets and pins are labeled (J1 and J2). The cable should point away from the Development Board, not across it.

3. The Personality Board has one jumper. Leave J8 open to use bus power to power the Application Board's circuits. Jumper J8 to power the Application Board from the Development Board's supply, with a limit of 100 milliamperes.

4. Plug the other end of the cable assembly into a Target Adapter. J3 and J4 on the cable assembly are two 40-pin sockets that mate with pins on one of the provided Target Adapters. The Application Board uses the 24P DIP Adapter. These connectors are keyed.

5. Plug the Target Adapter's pins into the DIP socket on the Application Board. This connection is not keyed. The cable should point away from the Application Board, not lie across it.

6. Connect an RS-232 cable from the Development Board to your PC's serial port.

7. Connect a USB cable from the *Application Board's* USB connector to a USB port on your PC. *Don't* use the USB connector on the larger Development Board.

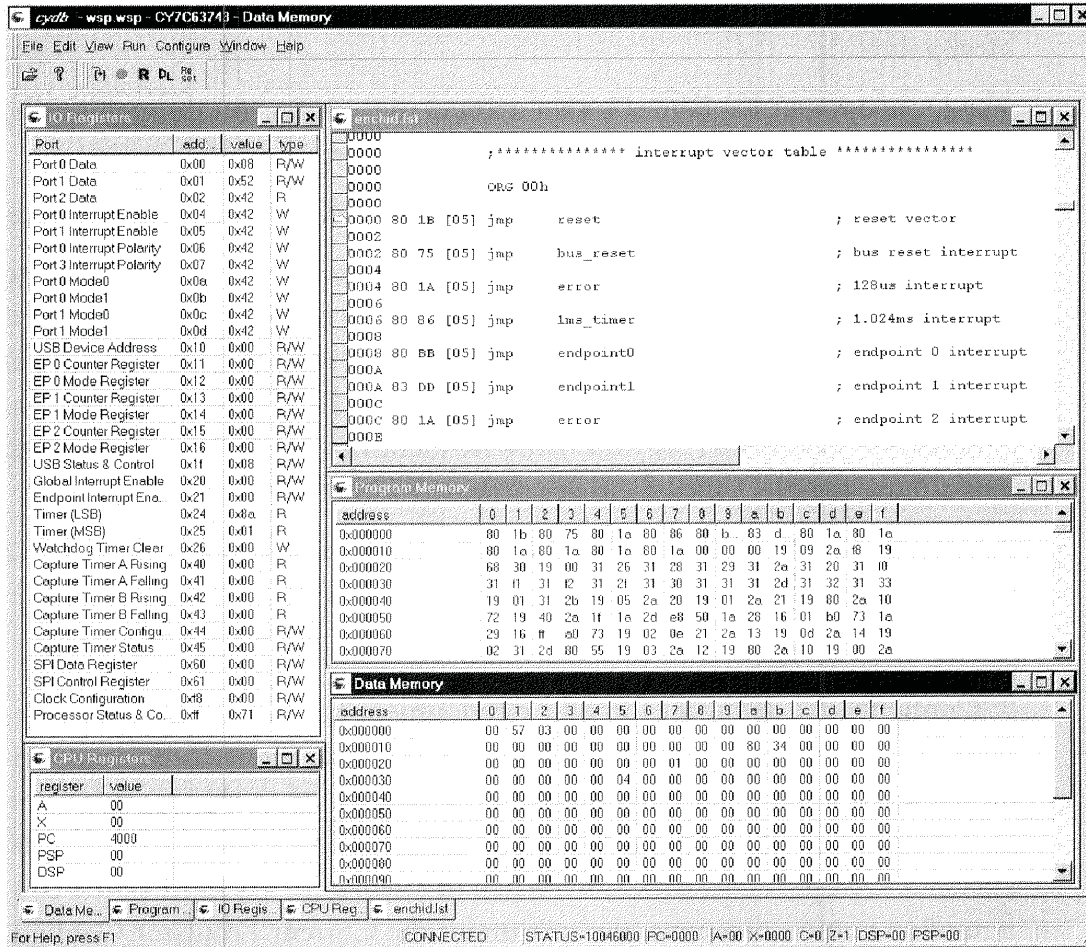8. Plug the power supply into an AC outlet and the Development Board's connector.

Figure 9-2: Cypress' CYDB monitor and debugger enables you to control program execution and view the status of memory and registers.

The Debugger

The companion to the development board is the CYDB debugger, or monitor program. In addition to enabling you to load and run your firmware, the debugger has features that can help enormously in tracking down program bugs.

Figure 9-2 shows the user screen, which you can customize to show the information you want. The View menu allows you to select which windows

display, including program and data memory, CPU and I/O registers, and breakpoints.

The Development Kit comes with a manual that guides you through setting up the system and getting started with the debugger.

Here's an example of how to use the Development Kit to run your firmware:

1. Write your source file in assembly code and use the Cyasm assembler to create an object file. The object file can be a *.rom* or *.hex* file, and contains your firmware's machine-code instructions in an ASCII Hex format. For your device to enumerate, it will also need an INF file on the host, as described in Chapter 11. If your firmware identifies the device as HID class, you can use the HID INF file that's provided with Windows.

3. Plug in the Development Board's power supply and connect the RS-232 and USB cables to the host PC.

4. Run the debugger.

5. Configure the debugger for your development hardware. From the Configure menu, select Target to display the Configure Target/Emulator window. Figure 9-3 shows the window as it appears after the configuration process is complete. Click the Connect button. In the window that appears, select a COM port and click OK. When the debugger has finished the configuration communications, the text under the Current Emulator Configuration label changes from Not Connected to Connected, and the Connect button's caption changes to Update. Click OK to close the window.

6. Download and run your code. To download code to the emulator, click the *DL* button or select Run, Download from the menu. In the window that appears, select a *.hex* or *.rom* file and a listing file and click OK. The debugger loads the selected file into the emulator's memory and displays the selected listing file.

To run the firmware, click the *R* button or select Run, Run from the menu. If all is well, the firmware will run and Windows will enumerate the device. The *R* button will be grayed out and the *Stop* button will appear as a solid red circle.
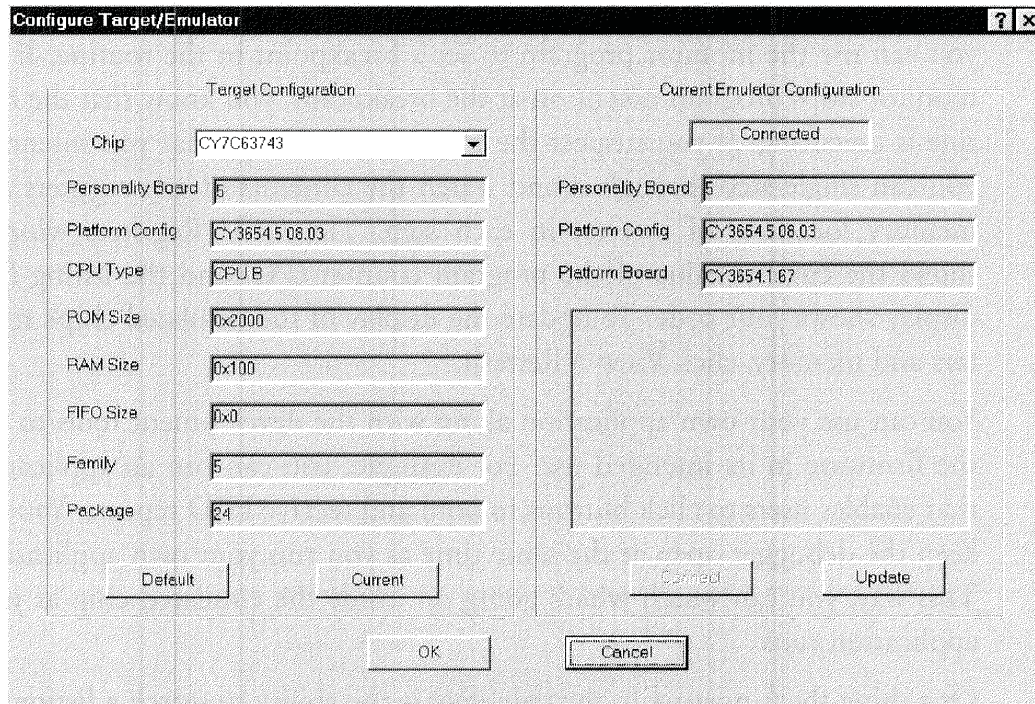
Figure 9-3: In the CYDB debugger, use the Configure Target/Emulator window to establish communications with the development board.

To stop the code, click the *Stop* button or click Run, Stop in the menu. To restart at the instruction where the firmware stopped, click *Run*. To restart from the beginning of program memory, click *Reset*.

Debugging Tips

The debugger enables you to precisely monitor and what the device's firmware is doing.

You can execute a portion of your application, then examine the states of all of the device's registers and RAM, or even change their contents on the fly. You can set a breakpoint to find out when and if a section of code executes. You can single-step through the code to find out exactly what the code does and where it branches. The Platform board's hardware and firmware disable the Watch Dog timer during single-stepping.

For example, if you suspect that a routine in your firmware never executes, you can use the monitor program to set a breakpoint in the routine. If the monitor stops program execution at the breakpoint, you know that the routine is executing. If you suspect the routine isn't doing what you intended, you can single-step through it and watch the contents of any registers and memory locations of interest in each step. The CPU Registers window shows the current value of the program counter (PC) and the listing file's display shows your code. To update the display of the emulated chip's registers and memory, click View > Refresh.

You can use your own application along with the development tools to test the firmware in its intended use. For example, you can run an application that enables users to click buttons to send and receive HID reports. You can keep the debugger open at the same time as you run your own application. This way, you can watch what's going on inside the emulated chip as your application runs.

One thing that's missing in the debugger is the ability to search a listing file for specific text. This makes it hard to find a specific line of code to set a breakpoint. So I keep a copy of the listing file loaded into a word processor and use that for searching. When I find the line of code I'm looking for, I note the line number and switch back to the debugger to set the breakpoint.

PROM Programming

When your code looks OK on the emulator and you're ready to try it out in a chip's PROM, you'll need a PROM programmer. Several vendors have programmers that are capable of this. An inexpensive one is the CY3649 Hi-Lo PROM Programmer, available from Cypress.

Programming chips in the enCoRe series requires two additional components, the CY3083-DP48 Adapter Base, which adapts the programmer for a specific package type, and the CY3083-08 Matrix Card, which routes the signals for a specific pinout. Both are available from Cypress.

Figure 9-4 shows the programmer, and Figure 9-5 shows the programmer application's display. The programmer is the same one provided with some
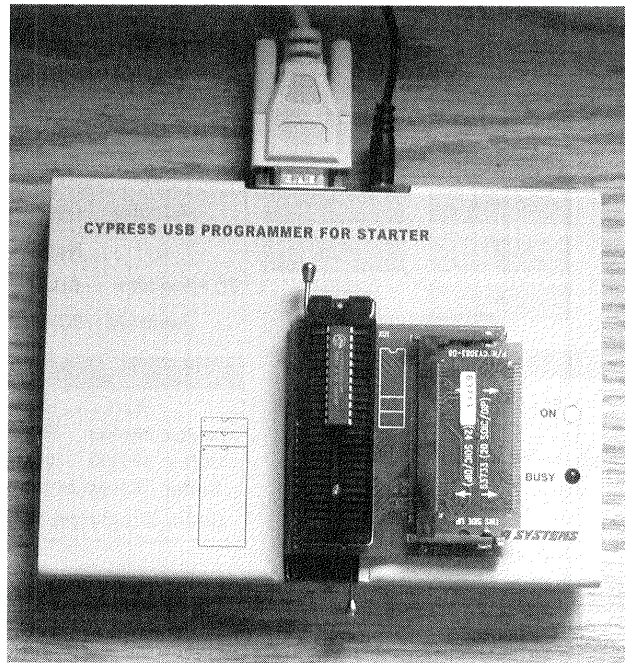
Figure 9-4: Cypress offers an inexpensive programmer and adapters for the enCoRe series and other chips. The photo shows an Adapter Base inserted into the programmer's ZIF socket. The Adapter Base holds a matrix card and a chip to be programmed.

of the now discontinued Starter Kits for the CY7C63000 series. If the programmer is labeled "Programmer for Starter," it's usable with the enCoRes if you update the software and get the Adapter Base and Matrix Card. If the programmer is labeled "Programmer for CY630...," it won't work with the enCoRes.

The programmer connects to the PC via an RS-232 serial port. (The unit was probably adapted from an existing design that predates USB.) As with other EPROM programmers, you place the chip to be programmed in a zero-insertion-force (ZIF) socket and flip the lever to lock in the chip.

These are the steps to program a chip:

1. Insert the Matrix Card into the Adapter Base and place the Adapter Base into the programmer's ZIF socket and lock it into place
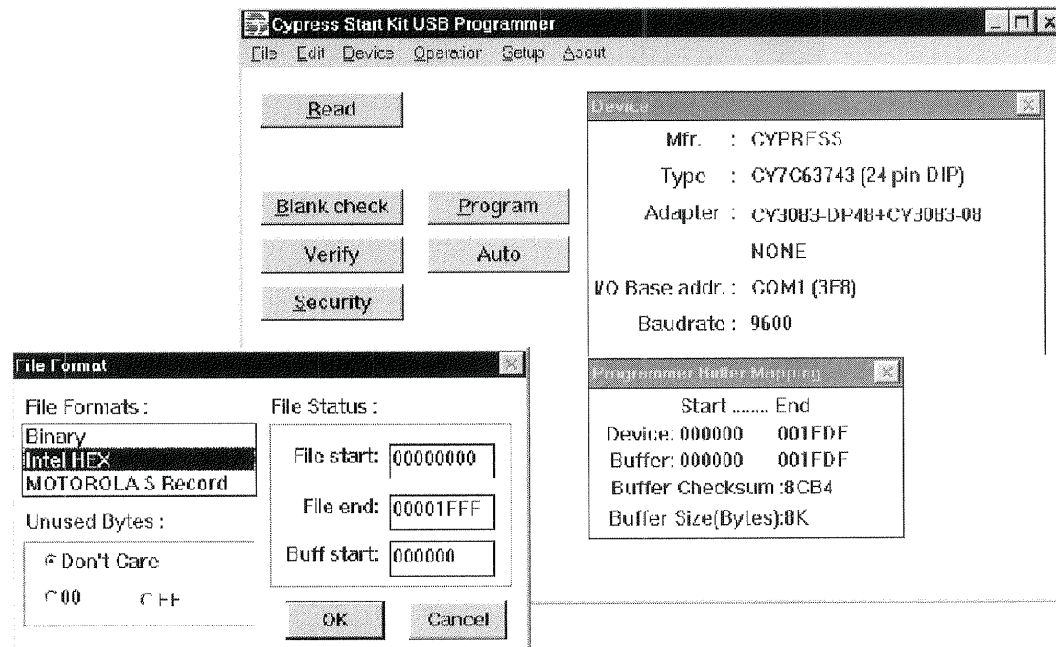
Figure 9-5: The software for Cypress Semiconductor's EPROM programmer enables you to program a file in any of several formats into Cypress chips, verify, and protect the code from copying by blowing the security fuse.

2. Place a chip to be programmed into the Adapter Base's ZIF socket and lock it into place.

3. In the Setup window, select a COM port and bit rate. A message will inform you when the software has located the programmer.

4. From the Device menu, select the device to be programmed.

5. From the File menu, select Load File to Buffer. Select a *.hex* file created by the Cyasm assembler. The programmer software is 16-bit, so long file and folder names will be truncated. In the window that appears, select file format = Intel Hex, File Start = 0000, File End = 1FFF, Buffer Start = 0000, and Unused Bytes = Don't Care.

6. Click Auto, then OK. This will cause the programmer to do four things in sequence. The programmer will verify that the chip is erased (contains all

FFs). It will program the buffer's file into the PROM, beginning at 0000h. It will verify that the chip's contents match the buffer.

The Security button blows the chip's security fuse to prevent anyone from reading the code stored in the chip. Anyone who tries to read the code in the device will see only FFs. Once the security fuse is blown, the device can no longer be programmed.

You can also do an individual blank check, program, verify, and security protection of the code. An edit menu enables you to edit individual bytes in buffer, search, move blocks of bytes, and fill areas with a value.

I found the programming software to be a little quirky. At higher bit rates, the programmer sometimes failed to read or program the device. After switching to 9600 bps, a device that failed at a higher bit rate passed the blank test but refused to be programmed until I re-erased. At slower rates, I had no problems. Because the amount to be programmed is small, the programming completes quickly enough even at a slower bit rate.

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 both with the system's USB drivers and with 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, data-acquisition unit, or just about anything controlled by circuits that the CPU can access. 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 a keyboard or mouse). The device may be a standard peripheral type or a unique design for a special purpose. It may be a one-of-a-kind, custom device. Some device drivers are class drivers that handle communications with a variety of devices that have similar functions.

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 that users run, including everything from popular word processors and databases to special-purpose applications that support custom hardware.

A device driver can enable application code to access a peripheral when the application knows only the peripheral's name (such as HP LaserJet) or the device's function (joystick). 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, and so on). Applications don't even have to know whether a device uses USB or another interface. The application code can be the same for all interfaces, with the hardware-specific details handled at a lower level.

A device driver accomplishes its mission by translating between application-level and hardware-specific code. The application-level code uses functions supported by the operating system to communicate with device drivers. 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 applications to communicate with device drivers. Applications written in Visual Basic, C/C++, and Delphi can call API functions. Three functions that device drivers may support for reading and writing to USB devices are ReadFile, WriteFile, and DeviceIoControl.

Although API functions simplify the process of communicating with hardware, they tend to have specific and rigid requirements for the values they pass and return. It's not unusual for a mistake in an API call to result in an application or even a 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 devices that connect to RS-232 serial ports. The controls provide an easier and more failsafe programming interface for setting parameters and exchanging data. The underlying code within the control may use API functions to communicate with device drivers, but the control insulates application programmers from dealing with the sometimes arcane details of the API calls.

Visual Basic doesn't have a generic control for USB communications, however. How an application communicates with a USB device varies with the driver assigned to the device. For example, a Visual-Basic application can use the Printer object to communicate with a USB printer.

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 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 function driver that manages communications between applications and the lower-level bus drivers. The bottom layer contains a bus driver that manages communications between the function driver and the hardware. One or more filter drivers may supplement the function and bus drivers.

The layered driver model is more complicated as a whole, but it actually simplifies the job of writing drivers. Devices can share code for tasks they have in common. Plus, the drivers that handle communications with the system's USB hardware are built into Windows, so driver writers don't have to provide them. Writing a device driver for a USB device is typically much

easier than writing a driver that has to handle the details of accessing the hardware.

Options for USB Devices

There are several approaches to obtaining a driver for a device. Sometimes you can use a driver that's included with Windows or provided by a chip vendor or other source. For other devices, you may need to write a custom driver. A variety of 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 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. All of these are available with a choice of interfaces, including USB. For example, a keyboard may use the original legacy keyboard interface or USB. A disk drive may use any of a number of interfaces, including ATAPI, SCSI, printer-port, IEEE-1394, and USB.

Windows includes class drivers for many standard device types. When devices in a class may have different interfaces, supplemental drivers can support the various interface options. And if a device has features or capabilities beyond what the class driver supports, a device-specific filter driver can support these as needed.

Custom Devices

Some peripherals are custom devices intended for use only with specific applications. Examples include data-acquisition units, motor controllers, and test instruments. Windows has no knowledge of these devices, so it has no built-in drivers for them. Devices like these may use custom drivers, or they may be designed so they comply with the requirements for a supported class. For example, a data-acquisition device may be able to use the HID drivers.

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. 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 it allows to applications.

User and Kernel Modes

Under Windows, code 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 USB device may also have a supplementary user-mode driver.

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. On Pentiums and other 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. On Pentiums and other x86 processors, kernel mode corresponds to the CPU's Ring 0 mode.
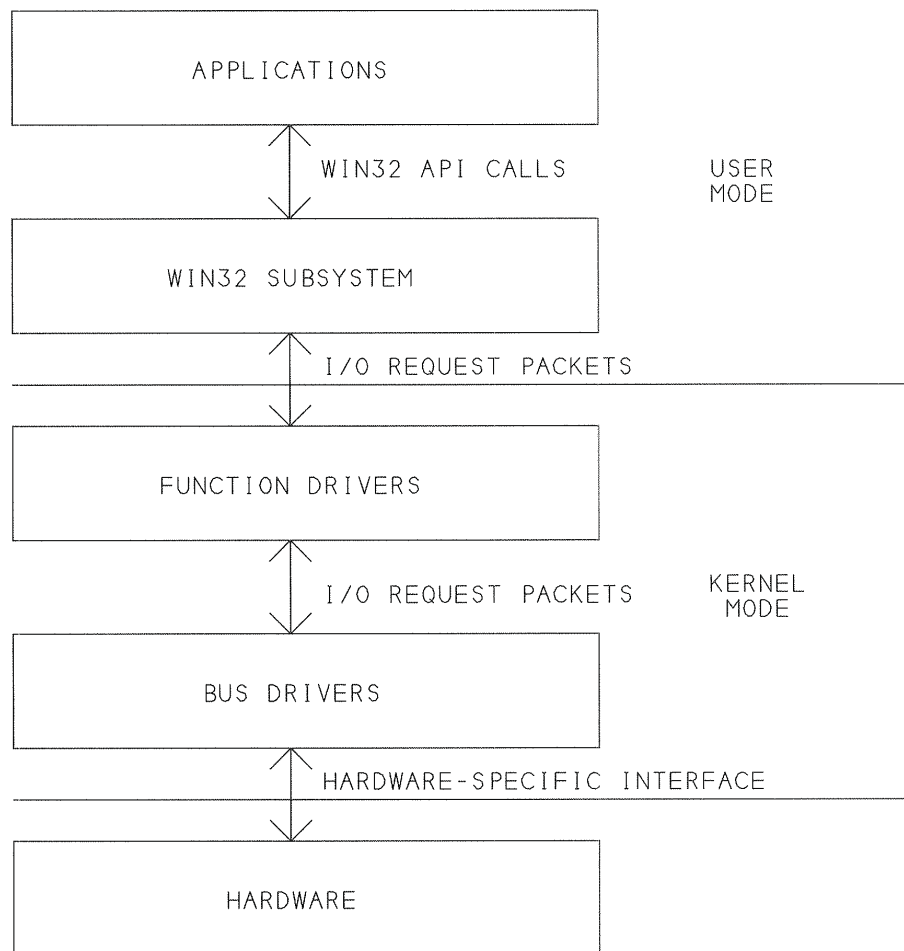
```
┌─────────────────────────────────────────┐
│                                         │
│            APPLICATIONS                 │
│                                         │
└─────────────────────────────────────────┘
                   ↑
                   │  WIN32 API CALLS         USER
                   ↓                          MODE
┌─────────────────────────────────────────┐
│                                         │
│         WIN32 SUBSYSTEM                  │
│                                         │
└─────────────────────────────────────────┘
                   ↑
          ─────────┼── I/O REQUEST PACKETS ──────────────
                   ↓
┌─────────────────────────────────────────┐
│                                         │
│         FUNCTION DRIVERS                 │
│                                         │
└─────────────────────────────────────────┘
                   ↑
                   │  I/O REQUEST PACKETS     KERNEL
                   ↓                          MODE
┌─────────────────────────────────────────┐
│                                         │
│           BUS DRIVERS                    │
│                                         │
└─────────────────────────────────────────┘
                   ↑
          ─────────┼── HARDWARE-SPECIFIC INTERFACE ──────
                   ↓
┌─────────────────────────────────────────┐
│                                         │
│             HARDWARE                     │
│                                         │
└─────────────────────────────────────────┘
```

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

Under Windows 98 and Me, applications can access I/O ports directly, unless a low-level driver has reserved the port, preventing access. Under Windows NT and 2000, only kernel-mode drivers can access I/O ports.

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 a single input or output action. A function driver for a USB device uses IRPs to pass communications to and from 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.

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, including Windows 2000 and Me. 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 the driver communicates with the operating system at a lower, 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 provides a common driver model for use by any device under Windows 98 and later. Earlier versions of Windows used different models for device drivers. Windows 95 used VxDs (virtual device drivers). Windows NT 4 used a type of driver called kernel-mode drivers. Developers who wanted to support both Windows 95 and Windows NT had to provide a driver for each. But a single WDM driver can work under both Windows 98 and Windows 2000.

The USB bus drivers included with Windows are WDM drivers. Although Windows 98 continues to support VxDs, USB devices must have WDM function drivers because their function drivers must communicate with the WDM bus drivers.

The Win32 Driver Model isn't completely new, but was built on existing components. A WDM driver is basically 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. These editions weren't available to retail customers, but were available only to vendors who installed the software on the computers they sold. Beginning with 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++. But to write a driver for a USB device, you need a tool that is capable of compiling a WDM driver, and this means using Visual C++. The exception is driver toolkits that provide a generic driver and either require no programming at all or permit you to use other C compilers or Delphi 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 it enables different devices that have tasks in common to 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 communications that are common to all. Including these drivers with Windows means that device vendors don't have to provide them. 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. Applications are at the top of the stack, and the USB hardware is at the bottom of the stack.

The Function Driver

A function driver enables applications to talk to a USB device 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, 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.

The function driver also knows how to communicate with the lower-level bus drivers that control the hardware. Figure 10-2 shows how these work together in USB communications. The function driver is often referred to as the device driver, though a complete device driver actually encompasses both the function driver and bus drivers. The function driver may be a class driver or a device-specific driver.

When a device or subclass has requirements beyond what a class driver handles, a supplemental driver called a filter driver can add the needed capabilities. An upper filter driver resides above the class driver. Requests from
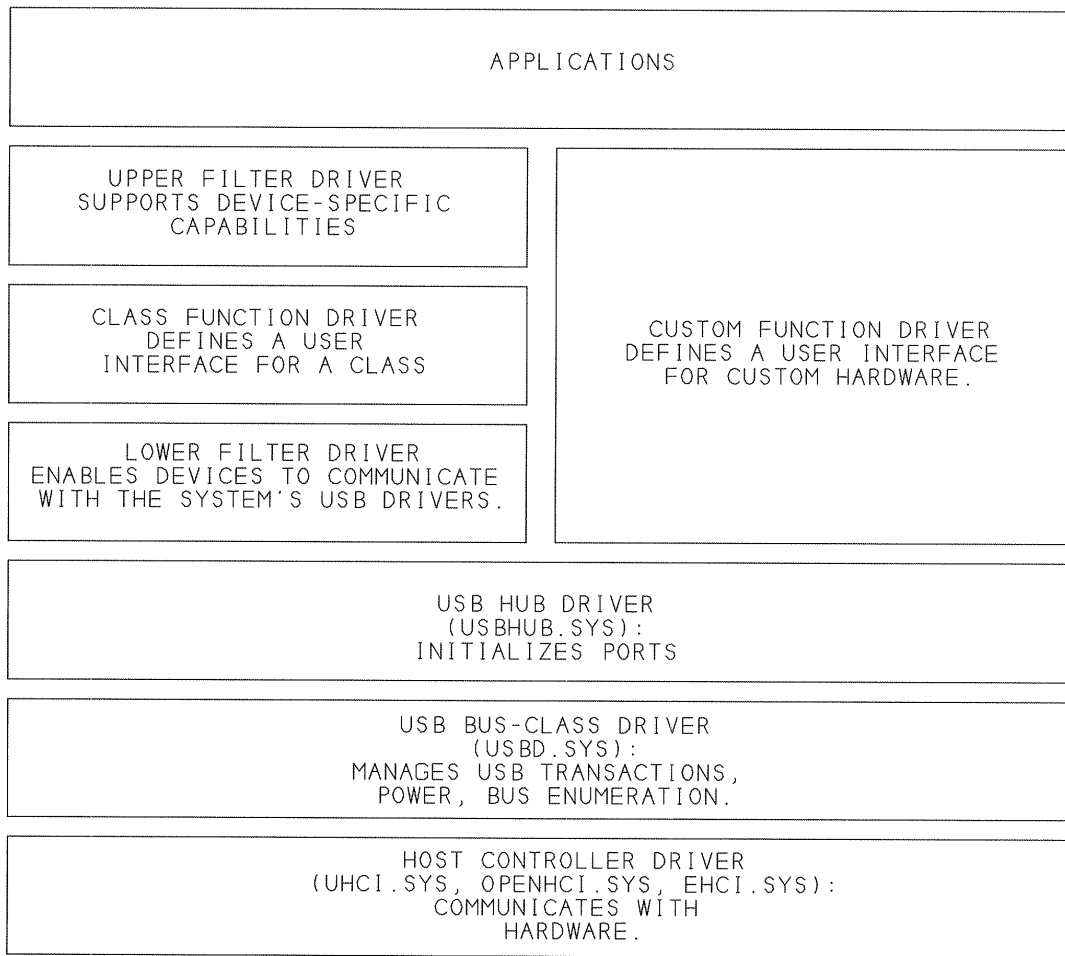
```
┌─────────────────────────────────────────────────────────────────┐
│                          APPLICATIONS                            │
│                                                                  │
└─────────────────────────────────────────────────────────────────┘

┌──────────────────────────────┐  ┌─────────────────────────────┐
│     UPPER FILTER DRIVER      │  │                             │
│   SUPPORTS DEVICE-SPECIFIC   │  │                             │
│         CAPABILITIES         │  │                             │
└──────────────────────────────┘  │                             │
                                   │    CUSTOM FUNCTION DRIVER   │
┌──────────────────────────────┐  │  DEFINES A USER INTERFACE   │
│    CLASS FUNCTION DRIVER     │  │     FOR CUSTOM HARDWARE.     │
│        DEFINES A USER        │  │                             │
│     INTERFACE FOR A CLASS    │  │                             │
└──────────────────────────────┘  │                             │
                                   │                             │
┌──────────────────────────────┐  │                             │
│      LOWER FILTER DRIVER     │  │                             │
│  ENABLES DEVICES TO COMMUNICATE │                             │
│  WITH THE SYSTEM'S USB DRIVERS. │                             │
└──────────────────────────────┘  └─────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐
│                        USB HUB DRIVER                            │
│                        (USBHUB.SYS):                             │
│                       INITIALIZES PORTS                          │
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐
│                     USB BUS-CLASS DRIVER                         │
│                         (USBD.SYS):                              │
│                   MANAGES USB TRANSACTIONS,                      │
│                    POWER, BUS ENUMERATION.                       │
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐
│                   HOST CONTROLLER DRIVER                         │
│          (UHCI.SYS, OPENHCI.SYS, EHCI.SYS):                      │
│                      COMMUNICATES WITH                           │
│                         HARDWARE.                                │
└─────────────────────────────────────────────────────────────────┘
```

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

applications pass through the upper filter driver before being passed to the class driver. A lower filter driver resides between the class driver and bus drivers. A class driver may pass requests to a lower filter driver, which in turn passes them to a bus driver. Lower filter drivers can enable a single class driver to support multiple interfaces, with each driver supporting the class-specific operations required for an interface. For example, Windows provides a driver that enables the HID-class driver to communicate with the USB bus drivers.

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's interface. The legacy virtualization driver is a VxD that knows how to talk to the HID driver.

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 manages bus power, enumeration, USB transactions, and communications between the root-hub driver and the host-controller driver. The host-controller driver enables the host controller hardware to communicate with the USB system software. The host controller connects to the bus. The host-controller driver is separate from the bus-class driver because Windows supports multiple types of host controllers, each with its own driver.

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. If you want to know more about how the low-level communications work, one source of information is the source code and other documentation from the Linux USB Project.

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 custom function 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.

When a device is attached, Windows' Device Manager handles enumeration automatically, 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 an instance of 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. The object may be any 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 may request a transfer. To read data from a data-acquisition unit, the 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

The Windows API includes three functions for exchanging data with devices: ReadFile, WriteFile, and DeviceIoControl. A driver may support any combination of these. 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 these functions.

Although the names suggest that they're used only with files, WriteFile and ReadFile are general-purpose functions that can transfer data to and from any driver that supports them. The data read or data to be written is stored in a buffer specified by the call. A call to ReadFile doesn't necessarily cause the driver to retrieve data from the device. The call may instead return data that was requested previously and stored in a buffer. The details vary with the driver. Chapter 15 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:

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

Windows may support additional API functions for transferring data with devices in a particular class. For example, the functions Hid_GetFeature and HidD_SetFeature read and send Feature reports to HID-class devices.

The Device 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 function 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 configuring devices and transferring data. The URBs are documented in the Windows DDK.

A function 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 transaction is skipped.

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

If you're using an existing function 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 function 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 includes the hub driver *usbhub.sys*.

The Bus-class Driver's Role

The USB bus-class driver translates communication 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 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.

There are three types of host controllers. Two are for low- and full-speed communications only and one is for high-speed communications only. The low- and full-speed controller types are the Open Host Controller Interface (OHCI) and Universal Host Controller Interface (UHCI). High-speed controllers must use the Enhanced Host Controller Interface (EHCI). The USB Implementers Forum's website has links to the specifications.

Controllers that conform to the OHCI standard use the driver *openhci.sys*, and controllers that conform to the UHCI standard use the driver *uhci.sys*. Both drivers provide a way for the USB hardware to communicate with the bus-class driver. Although they differ in how they do so, in most cases the differences are transparent to driver developers and application programmers.

The two drivers take different approaches to implementing the host-controller's functions. UHCI places more of the communications burden on software and allows the use 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 two host controller types do have some differences in performance. 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, the Windows UHCI driver attempts no more than one transaction per frame, while an OHCI driver may schedule additional transactions in a frame. And 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.

An EHCI controller handles high-speed communications only. To support all three speeds, a PC must have an EHCI controller and either a companion OHCI or UHCI controller in the PC or a 2.0-compliant hub, which performs the function of a host controller for low- and full-speed devices. An EHCI host controller and a companion 1.x host controller can share a single bus. Users and application programmers don't have to know or care which host controller is communicating with a device.

The Device's Role

After a transmission leaves 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

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 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.

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 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 driver causes 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 make periodic requests for keypress data because there is no way for an application to predict when a key will be pressed.

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 resuming communications.

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, the easiest approach to accessing a USB device is to use a driver included with Windows. This way, there are no drivers to write or install and any Windows computer can access the device. Chapter 12 has details about the class drivers available in Windows. For custom designs, the most useful of these are the HID drivers and possibly the mass-storage driver.

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 open a handle to the device and read and write to it in application code. The usefulness of vendor-supplied drivers varies. A driver is much less useful if it's buggy, doesn't include the features you need, or has sketchy documentation that makes it hard to understand and use.

Chapter 12 describes drivers from FTDI for use with its USB UART chip and from SigmaTel for use with its IrDA-to-USB bridge chip

Custom Drivers

Sometimes there is no generic or vendor driver that includes the transfer types you want to use or has the performance you need. Or you may want to define custom DeviceIoControl codes. In these cases, the solution is to create a custom device driver. The next section discusses this option.

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. On the positive side, writing a USB driver is easier than writing a driver for a device that connects to the ISA bus. Plus, a variety of products can help to simplify and speed up the process.

Requirements

The minimum requirement for writing a device driver from scratch 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 Device Developer's Kit (DDK), a subscription to Microsoft's Developer's Network (MSDN), driver toolkits, and advanced debuggers.

The Windows DDK includes example code and developer-level documentation. The USB-related documentation includes tutorials on WDM drivers and HIDs and source code for USB drivers.

For bulk transfers, the DDK includes source and compiled code, documentation, and an example application for the *bulkusb.sys* driver. The driver 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 DDK includes the *isousb.sys* driver for handling isochronous transfers. If you decide to use either of these, check the USB Implementers Forum's webboard for tips and fixes before you begin!

The DDK also has a filter-driver example and the *usbview* utility. The examples can be a useful starting point in developing your own drivers. You can download the Windows 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 and USB, 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.

How to write a USB driver from scratch is a much bigger topic than this book has room for. Some excellent books cover the topic in detail, including WDM device-driver writing in general as well as sections specifically about USB. Three good books are *Programming the Microsoft Windows Driver Model* by Walter Oney, *Writing Windows WDM Device Drivers* by Chris Cant, and *Developing Windows NT Device Drivers* by Edward N. Dekker and Joseph M. Newcomer. (NT drivers are similar to WDM drivers, and the book includes material on WDM and USB.) Chapter 17 describes Microsoft's programs for driver testing and digital signing.

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 BSQUARE, Jungo Ltd., and Compuware NuMega.

There are two general categories of toolkits. One provides a generic driver that handles USB communications, generates an INF file, and provides other assistance in enabling applications to use the driver. This approach is very fast and requires no programming at all to create the driver, but it 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.

Toolkits that Use a Generic Driver

All USB communications follow the protocols defined in the specification, so it makes sense that a single generic driver should be able to communicate with just about any device. A generic driver would have to support all four transfer types, including vendor-defined control requests, plus it should 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.

Two toolkits enable a device to use a generic driver: BSQUARE's WinRT for USB and Jungo's WinDriver USB. These toolkits require no driver programming at all.

WinRT for USB. WinRT for USB includes a kernel-mode driver and several supporting files. The driver supports synchronous and asynchronous transfers of all four types, retrieving descriptors and the device GUID, selecting an interface, and registering for device notification to detect when a device is removed from the bus. For example, to request an interrupt transfer, an application calls the function WinRTInterruptTransfer, passing the device handle, endpoint number, buffer length, and a buffer. The function returns a status code and the number of bytes transferred.

To create the files needed to support a device, you develop your device firmware, store the firmware in the device, and attach the device to the bus. To make the required setup files for the driver, run the WinRT for USB Con-
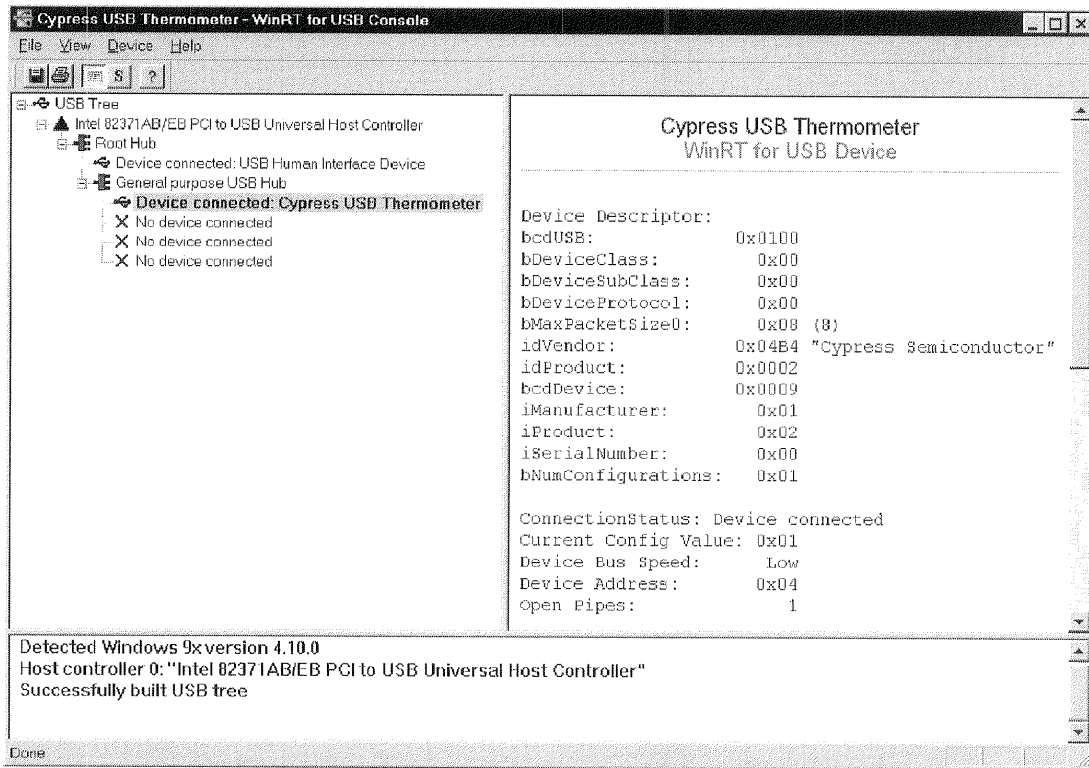
Figure 10-3: The WinRT USB console detects attached devices, displays descriptors, and creates a driver and the setup files for a device.

sole application (Figure 10-3) and select your device from the tree of detected USB devices. The Console prompts you for a symbolic name for your device, which can be anything you specify, and other optional information. The Console then makes the setup files and offers to install the driver on the current system. For testing, the WinRT for USB Wizard creates a sample Visual C++ application.

In addition to the driver file, there are two C header files containing the function prototypes and data types for calling the functions in the driver and error codes and *.dll* and *.lib* files that enable applications to access the functions in the driver. Chapter 15 has more about using *.dll* and *.lib* files.

When you distribute the device, you also distribute the INF file created by the Console application, *WinRTUsb.dll*, *WinRTUsb.sys*, and any application software you provide.

USB Complete

Applications can also access WinRT USB's functions from the provided ActiveX control. To enable using the control with Visual Basic, you add it to a project by clicking Project > Components > Controls and selecting the WinRT-USB control. The Object Browser then shows the supported classes and their properties, functions, and subroutines. This line of Visual-Basic code performs a bulk transfer:

```
returnlength = WinRTUsb1.BulkTransfer(0, size, buffer)
```

There are two editions of WinRT for USB. One is for use with Windows 98, Windows 2000, and Window Me. The other enables you to provide a driver for use on Windows NT 4.

WinDriver USB. Jungo's WinDriver USB takes a somewhat different approach but also can provide a driver without requiring you to write any code. The WinDriver Wizard generates files that you compile to create a custom user-mode driver in an *.exe* file. The user-mode driver communicates with the provided kernel-mode driver *windrvr.sys*. You can compile the files generated by the Wizard using Visual C++, C++ Builder, or Delphi. WinDriver will also create an INF file for the device.

The WinDriver Wizard enables you to select your device from those detected, then test it immediately by reading and writing data (Figure 10-4). You can then request the Wizard to create the driver files. When the driver is installed, applications 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 DebugMonitor application enables you to monitor activities handled by *windrvr.sys*. WinDriver USB's drivers run under Windows 98 and Windows 2000.

Toolkits that Provide Libraries for Creating a Custom Driver

The completely automated toolkits aren't suitable for every device. They can't create filter drivers, and you may want a completely custom driver to achieve the best possible performance. Three products for creating custom
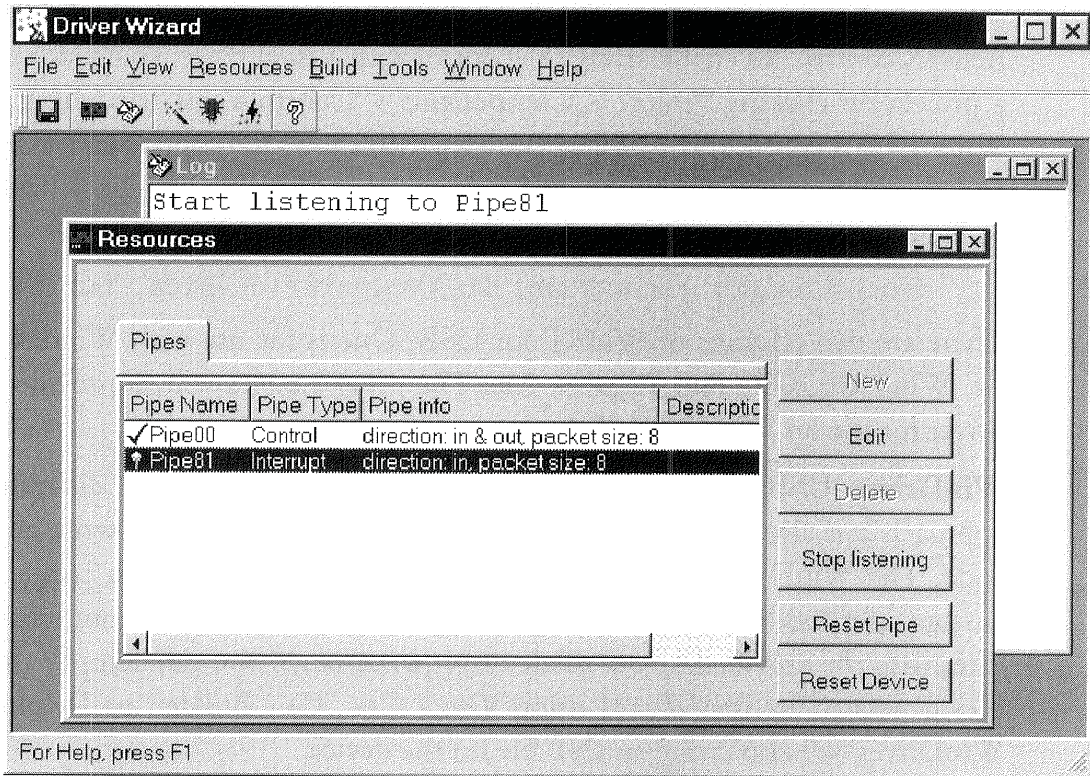
Figure 10-4: WinDriver's Driver WIzard enables you to test your device firmware by reading and writing to it, then creates the files you compile to create a custom driver for the device.

drivers are BSQUARE's WinDK, CompuWare Numega's DriverWorks, and Jungo's KernelDriver.

Each of these 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 is the same as if you had written the driver from scratch.

Each of these toolkits is capable of generating driver code for any device type, not just USB devices. WinDK has an optional USB extension that enables you to use the same source code to create a driver that will run on Windows NT 4.

11

How Windows Selects a Driver

When Windows detects a new USB peripheral, one of the things it has to do is decide which device driver applications should use to communicate with the device and if necessary, load the selected driver. This is the job of Windows' Device Manager, which uses class and device installers and INF files to find a match.

This chapter explains how these components work together to select drivers for newly attached devices. I also show how to create an INF file that will cause the Device Manager to select the correct drivers.

The Process

The Device Manager is a Control-Panel applet that's responsible for installing, configuring, and removing devices. The Device Manager also adds information about each device to the system registry, which is the database

that Windows maintains for storing critical information about the hardware and software installed on a system.

In Windows 98, display the Device Manager by right-clicking the My Computer icon on the desktop and selecting Properties, then the Device Manager tab. Or select Start Menu > Settings > Control Panel > System > Device Manager. In Windows 2000, it's the same except for one more click after System: System > Hardware > Device Manager.

The device and class installers are DLLs. Windows has default installers that the Device Manager uses to locate and load drivers for devices in the classes supported by the operating system (such as HIDs). The Device Manager and the installers together are also responsible for displaying dialog boxes as needed to prompt users for information.

The INF file is a text file containing information that helps Windows identify a device. The file tells Windows what driver or drivers to use and what information to store in the registry.

Searching for INF Files

When Windows enumerates a new USB device, the Device Manager compares the data in all of the system's INF files with the information in the descriptors retrieved from the device on enumerating. A typical PC can accumulate hundreds of INF files, so Windows 98 and Windows 2000 have ways to speed up the search.

To prevent having to read through all of the INF files each time a new device is detected, Windows 98 maintains a driver information database with information culled from its INF files. The database files are *drvdata.bin* and *drvidx.bin*, stored in the *windows\inf* folder.

You can view the contents of these files in a text editor or word processor. (Ignore the extra characters in the files.) Don't change the contents of the files, however; when you're finished viewing, just close the files without saving.

Drvidx.bin lists every Vendor and Product ID in the INF files, along with the manufacturer name, provider name, and description. *Drvdata.bin*

matches manufacturers with INF files that contain information about their products. After retrieving the Vendor and Product IDs from a device, the Device Manager uses the information in these two files to find the manufacturer and the INF file with information about the specific product.

Windows 2000 doesn't have these database files, but instead uses PNF (precompiled INF) files to speed searching. During device installation, Windows 2000 creates a PNF file and stores it in the same folder as the device's INF file. The PNF contains much of the same information as the INF but in a format that enables quicker searching. Windows 98 systems may have PNFs also.

The Registry's Role

The system registry stores information about all installed devices, whether or not they're attached and enumerated. When a new device is enumerated, the Device Manager stores information about the device in the registry.

To learn what kinds of information the Device Manager finds and stores, you can view (and edit) the registry's contents using the *regedit.exe* utility that comes with Windows.

A word of caution: the system registry is a vital and essential component of Windows. It's so important that Windows maintains multiple backup copies in case the current copy becomes unusable. Be extremely careful about making changes to the registry. If you goof and want to restore the registry to its previous state, boot to the DOS prompt and type *scanreg /restore.* Just viewing the registry is safe, however.

The registry arranges its contents in a tree structure. Information about USB devices is in a couple of places:

HKEY_LOCAL_MACHINE\Enum\USB

lists all USB devices.

HKEY indicates a registry key, which is an item in the registry structure. HKEY_LOCAL_MACHINE is a pointer to a data structure containing information about the system's hardware and installed software.

USB devices are also listed in this branch:

> HKEY_LOCAL_MACHINE\System\CurrentControlSet\
> Services\Class

The Class branch has sub-branches for various categories. The USB branch lists the USB host controller and root hub, as Figure 11-1 shows. A USB peripheral doesn't necessary show up in the USB branch; it may be in a branch that pertains to the peripheral's function. Standard peripheral types like keyboards, mice, and printers have their own branches, and will show up there. HID-class devices also have an entry in the HID branch. Other peripherals, such as digital cameras, may be in the USB branch. If the Device Manager can't figure out what to do with a device, it may call it an Unknown Device and place it in the USB branch. A custom peripheral can also create its own branch.

The Control Panel

The Device Manager is also responsible for adding attached devices to the Device Manager's window, as Figure 11-2 shows.

The Device Manager's display shows only the USB devices that are currently detected. You can unplug a device while viewing the display and watch the device's listing disappear. Plug the device back in, and its listing pops back. An exclamation point over the device's icon means that there was a problem communicating with the device or finding a driver. An X over the icon means that the device is present but disabled, possibly by the user. To view additional information about a device, select the device and click Properties.

What the User Sees

What you see on the screen when you attach a new USB peripheral depends on what drivers and INF file the device uses and whether or not the device has been attached and enumerated previously.
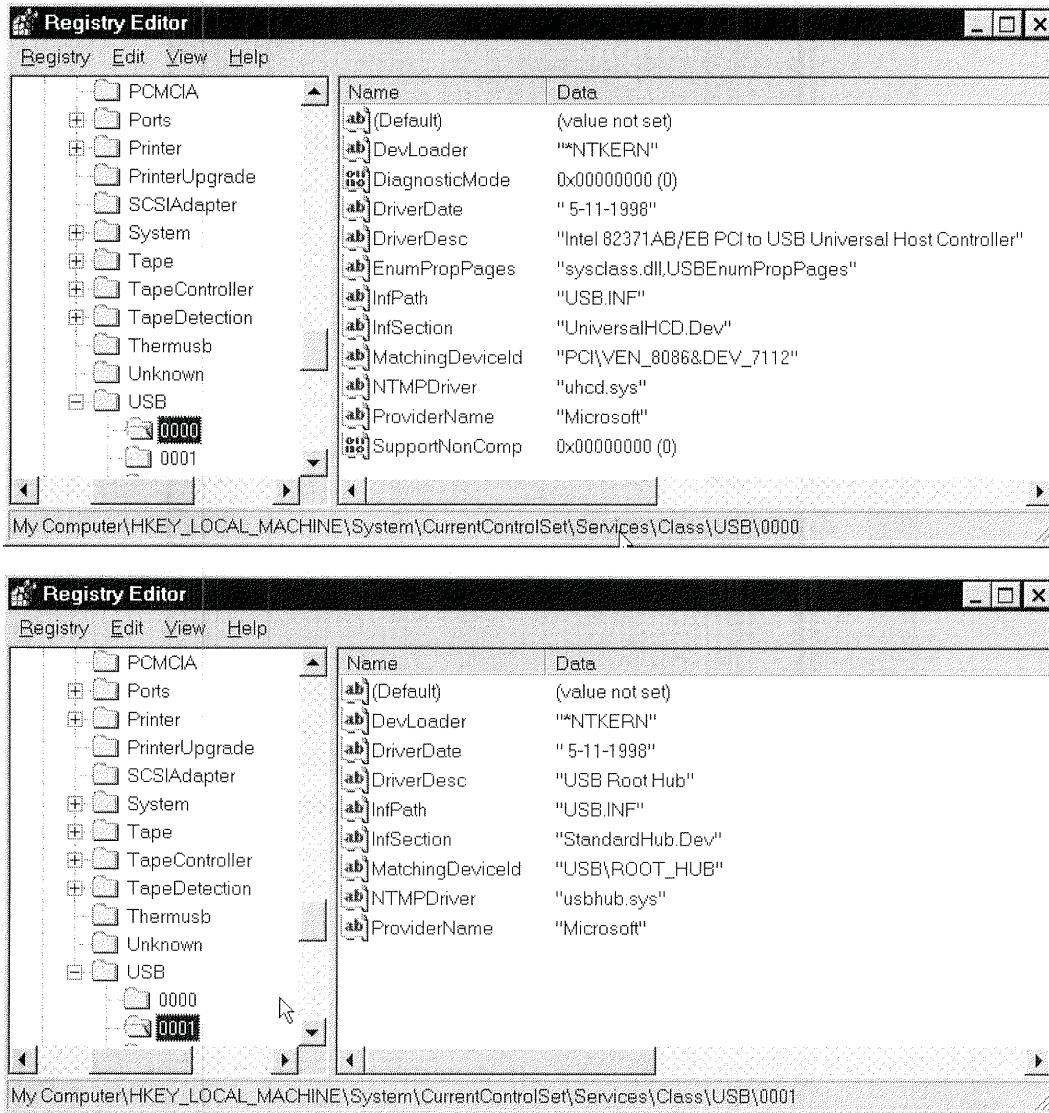
Figure 11-1: The registry's Class\USB branch has information about the system's host controller and root hub.

Specific Device Listings

When you attach a device, Windows displays a window with the message *New Hardware Found.* If the device descriptors include a Product String, under Windows 98 SE and later, the window displays the string. Otherwise,
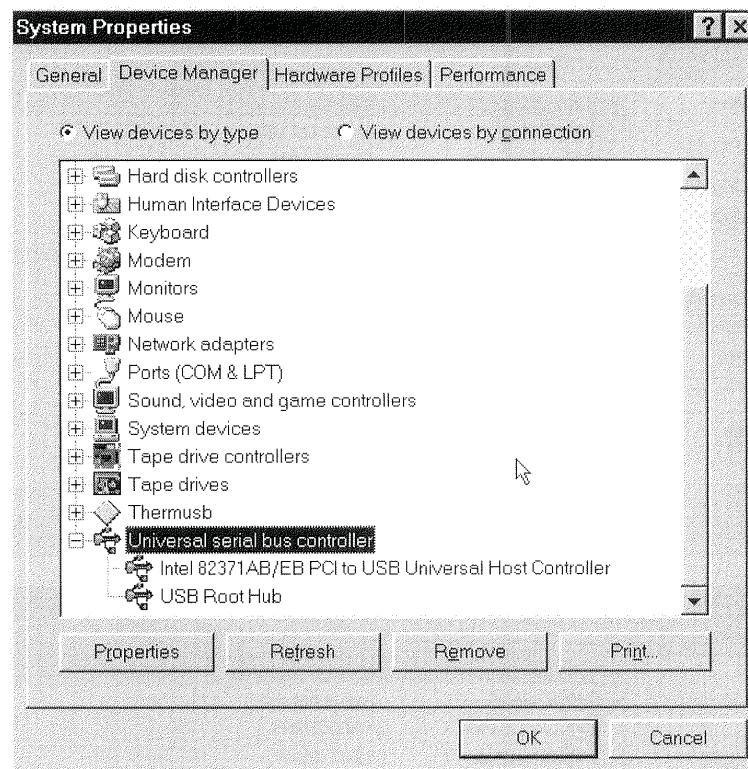
Figure 11-2: The Control Panel's Device Manager lists all attached and enumerated devices.

it displays *Unknown Device*. If the device has never been enumerated on the system, Windows will need to locate a driver.

If Windows doesn't find a matching INF file, it runs the Add New Hardware Wizard (Figure 11-3). You see a window recommending letting the Wizard search for the best driver for the device. When you accept the recommendation and select Next, the Wizard requests a location to search.

If the device comes with a driver on disk, specify the drive containing the disk. When the Wizard finds the file, it displays the filename and announces that it's ready to install the driver. (To make things as easy as possible for users, vendors should store the INF file in the root directory of the product's disk.) Click Next, and the Wizard displays *Please wait while Windows builds a driver information database.*

USB Complete
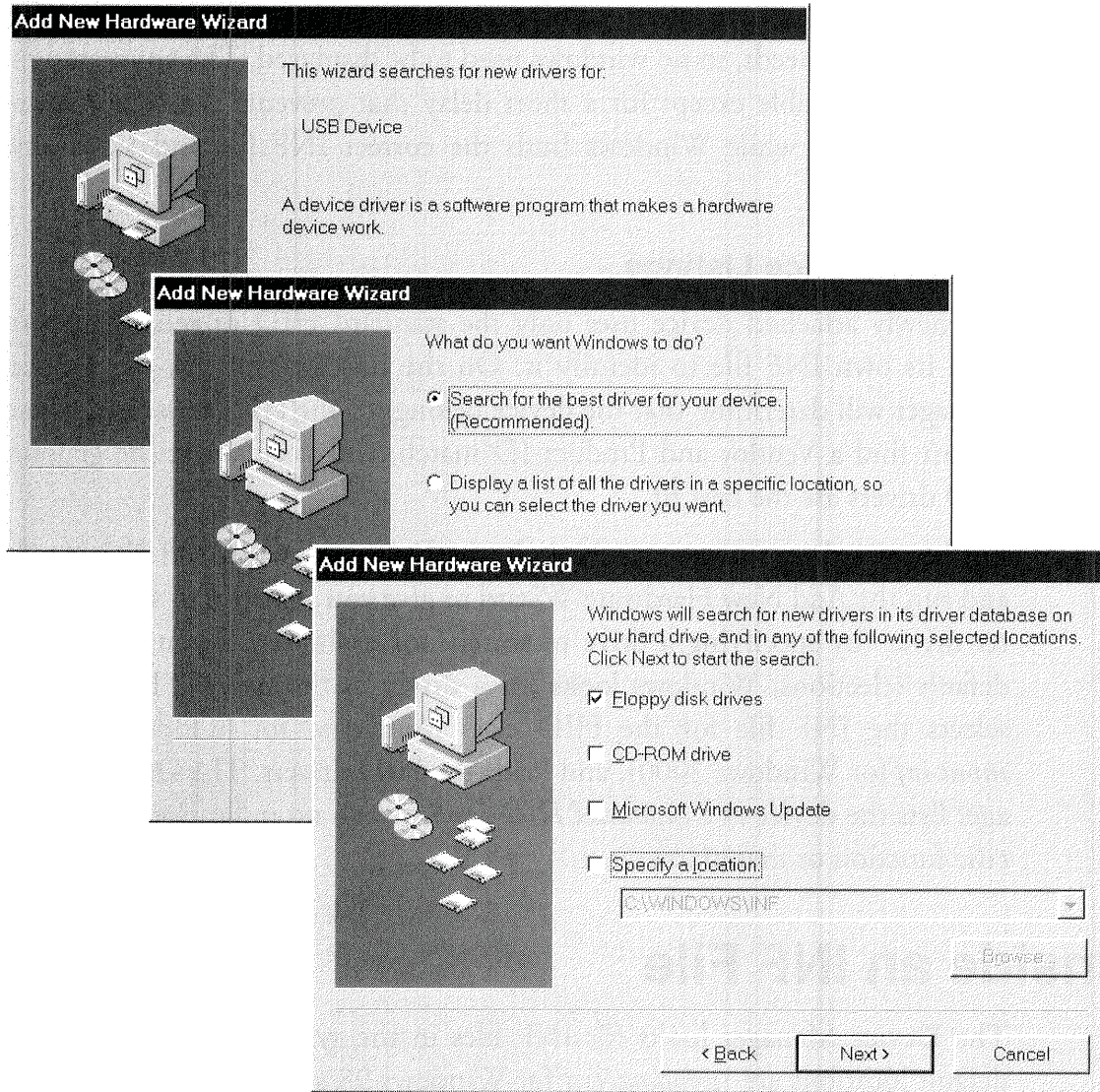
Figure 11-3: Windows' Add New Hardware Wizard searches for and installs drivers for newly attached devices.

The Wizard copies the INF file to the system's INF folder, loads the driver(s) specified in the file, lists the device in the Device Manager, and displays a window letting you know that it has finished installing the required software. The Device Manager's listing shows the device description, manufacturer, and provider name from the INF file.

If the device has been enumerated previously, the system already has the information it needs, so no windows need to be displayed. The enumeration should be invisible except for a short delay that prevents the cursor from selecting items while Windows finds the correct INF file and loads any needed drivers.

Generic Device Listings

If a newly attached device uses only the standard HID drivers, it doesn't need its own INF file to identify it. On the first attachment, the Device Manager will determine that the device belongs to the HID class, and when it can't find a Vendor and Product ID match, will decide that the generic HID drivers are the best fit.

But because there was no exact match, the Device Manager will play it safe and run the Add New Hardware Wizard to give you a chance to select a better driver (by specifying a drive to search, for example). If you accept the default selections, Windows looks for a driver in the system's INF folder, selects the INF file for the HID class (*hiddev.inf* for Windows 98 or *input.inf* for Windows 2000), and loads the HID drivers. The Device Manager lists the device as a *Standard HID Device*, with no indication of its specific function or manufacturer.

Inside an INF File

The Device Manager looks for INF files in the system's INF folder. The default locations are *windows**inf* for Windows 98 and *winnt**inf* for Windows 2000. By default, this is a hidden folder. If you don't see the folder in My Computer, select View > Folder Options > View, then under Hidden Files, select *Show all files*. Do *not* click *Hide file extensions for known file types*.

Examining the existing files is a good way to learn about the kinds of things contained in the files and how the information is structured. Your PC is sure to have plenty of INF files to examine. The INF file for the HID class is *hiddev.inf* in Windows 98 and *input.inf* in Windows 2000. INF files can be long and complicated, but the basics are fairly straightforward. In most cases, you can create an INF file by adapting one that's similar to what you

need. Vendors of USB controller chips often provide examples. The Windows DDK also has documentation on the contents and structure of INF files.

INF files for Windows 2000 have a few changes compared to Windows 98, including the need for a Services section that specifies how and when a driver's services are loaded. The DDK documentation has more details under *INF File Sections and Directives*.

Listing 11-1 is an INF file for a custom HID under Windows 98. I used *hiddev.inf* and Cypress' example INF files as models for the file. Figure 11-4 and Figure 11-5 show the information that the Device Manager displays after enumerating a device with this INF file.

Syntax

The information in an INF file must follow a few syntax rules, which will look familiar if you have experience with the *.ini* files commonly used in Windows 3.x.

- The information is arranged in sections, with each section containing one or more items. The section name is in square brackets [ ]. A carriage return/line feed begins a new item. Some of the section names (Version, ClassInstall) are standard names that Windows will look for. Other sections match values specified in other sections. For example, if the Manufacturer section designates the manufacturer as Lakeview, the INF file will also have a Lakeview section. The sections can be in any order, though most follow the same convention, and the order of the items within a section can be critical. So if you're adapting an example, keep the order of items in the sections the same.

- A semicolon (;) indicates a comment.

- A backslash (\) at the end of a line acts as a line continuator, unless it's enclosed in quotes ("\").

- Text enclosed in percent symbols (%sampletext%) refers to a string. For example, you might have the following item:

```
provider=%Provider%
```
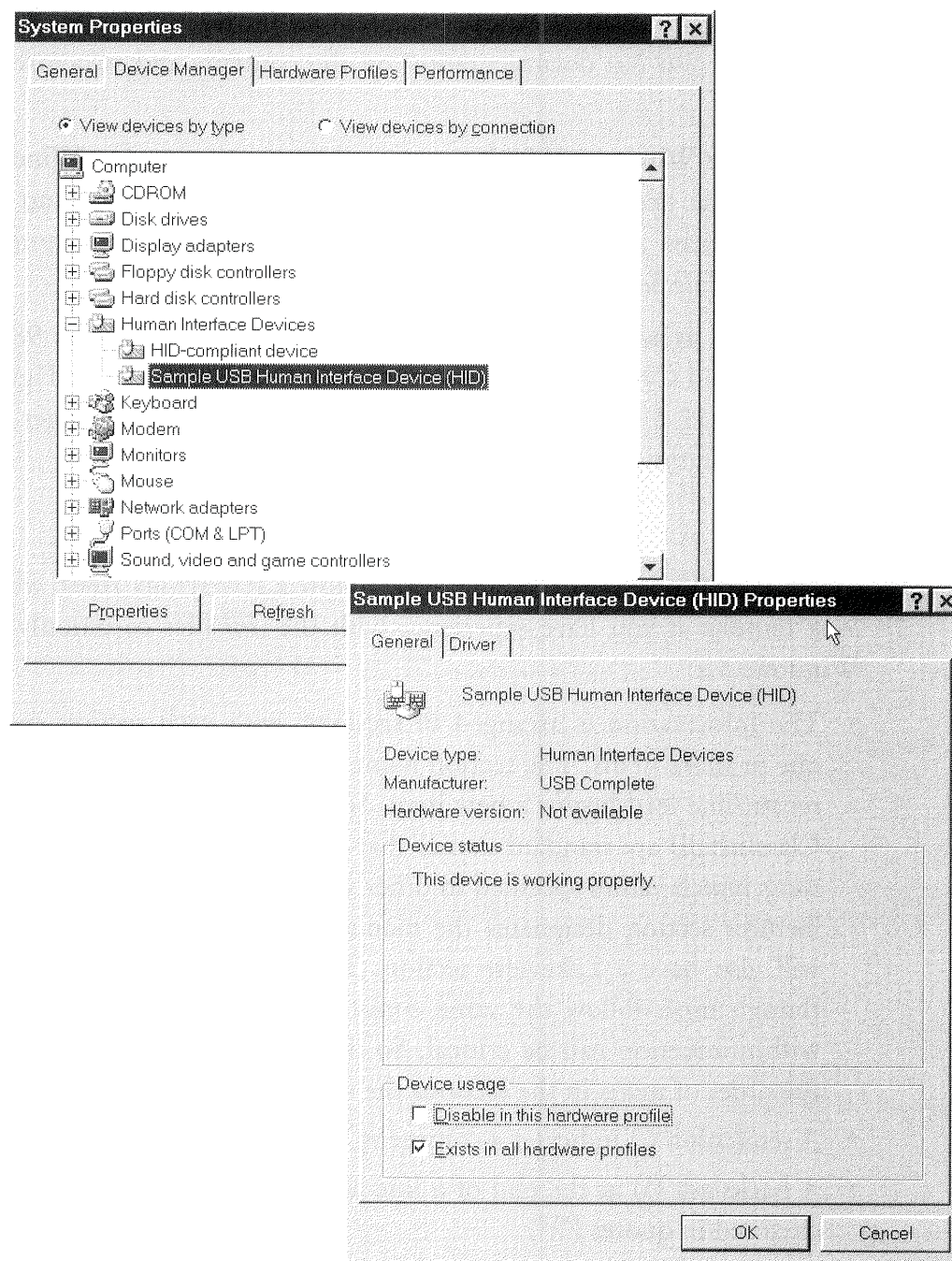
Figure 11-4: The Device Manager displays information obtained from the device's INF file. The device is listed both as an HID compliant device and as a device matching the description and Manufacturer in the INF file.
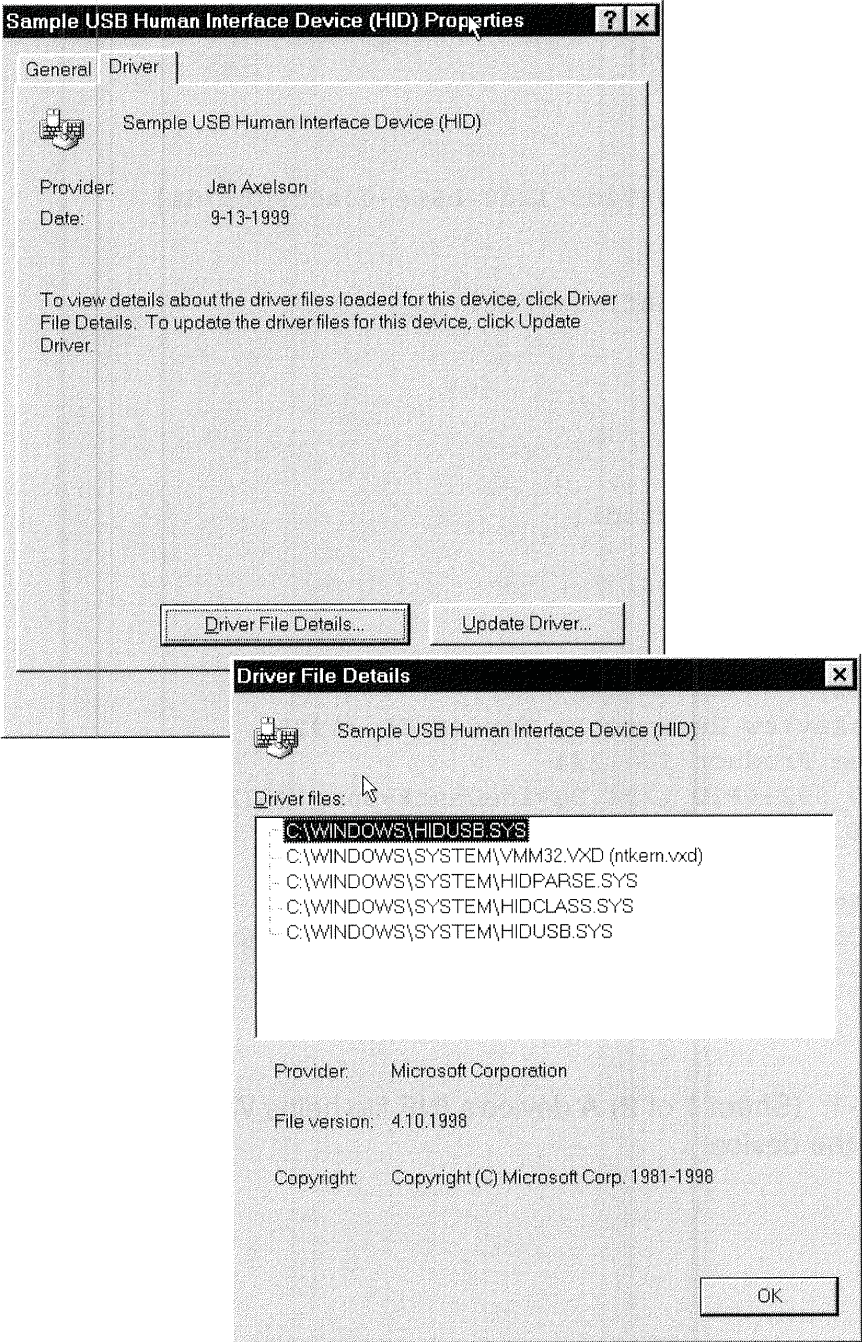
Figure 11-5: The information displayed by the Device Manager includes the Provider name and drivers specified in the device's INF file.

```
[Version]
Signature="$CHICAGO$"
Class=HID

;The GUID for HIDs
ClassGUID={745a17a0-74d3-11d0-b6fe-00a0c90f57da}

provider=%Provider%
LayoutFile=layout.inf, layout1.inf

[ClassInstall]
Addreg=Class.AddReg

[Class.AddReg]
HKR,,Installer,,mmci.dll

[Manufacturer]
%MfgName%=Lakeview

[Lakeview]
;Uses Lakeview Research's Vendor ID (0925)
;Uses the Product ID 1234
%USB\VID_0925&PID_1234.DeviceDesc%=SampleHID,
 USB\VID_0925&PID_1234

[DestinationDirs]
USBHID.CopyList = 11                ; LDID_SYS
;------------------------------------------------------------;
```

Listing 11-1: (Sheet 1 of 2) A device's INF file helps Windows locate the driver
to use for the device.

```
[SampleHID]
CopyFiles=SampleHID.CopyList
AddReg=SampleHID.AddReg

[SampleHID.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,"hidusb.sys"

[SampleHID.CopyList]
hidusb.sys
hidclass.sys
hidparse.sys
;----------------------------------------------------------------
 -;
[Strings]
Provider="Microsoft"
MfgName="USB Complete"
USB\VID_0925&PID_1234.DeviceDesc="Sample USB human interface
 device (HID)"
```

Listing 11-1: (Sheet 2 of 2) A device's INF file helps Windows locate the driver to use for the device.

with an item in the Strings section that defines the provider string:

```
Provider="USB Complete"
```

- Some items set the value of an entry. For example, this item defines the device's class entry as HID:

```
Class=HID
```

- Some items specify information to store in the system registry:

```
HKR,,Installer,,mmci.dll
```

Sections

An INF file includes sections that help Windows identify the device, find the appropriate drivers, and store information about the device in the system registry. Here is the purpose of each section in the example INF file:

Version

The Version section is the file's header. Every INF file must have one.

The Version section in the example file has these items:

```
[Version]
Signature="$CHICAGO$"
Class=HID
;The GUID for HIDs
ClassGUID={745a17a0-74d3-11d0-b6fe-00a0c90f57da}
provider=%Provider%
LayoutFile=layout.inf, layout1.inf
```

The Signature key specifies which operating system the INF file is intended for. For devices that use WDM drivers, the value can be $Windows 98$, $Windows NT$, or $Chicago$, no matter which operating system the PC is using. Chicago was a beta name used when Windows 95 was under development and its use is still valid under later editions of Windows.

The Class key specifies the class for devices installed with this file. The example specifies the HID class.

The ClassGUID key specifies the GUID in the registry for devices installed with this file. A GUID is a 128-bit identifier. The example is the GUID for the HID class. It uses the standard GUID format. There's more on GUIDs later in this chapter.

The Provider key names the creator of the INF file. In the example, %Provider% refers to a string defined later in the file.

The LayoutFile key names the source disks and files needed to install the driver for the device. Because the HID drivers are included with Windows, the example specifies files that contain installation information for the Windows setup. These files are also INF files. The information is in the SourceDisksFiles and SourceDisksNames sections of the files.

ClassInstall

The ClassInstall section installs a new class in the Class section of the registry. The Device Manager processes this section only if a device's class isn't yet installed in the operating system.

The example ClassInstall section has one item:

```
[ClassInstall]
Addreg=Class.AddReg
```

The Addreg key adds a class description to the system registry. In the example, the key's value refers to the Class.Addreg section, which specifies an installer file:

```
[Class.AddReg]
HKR,,Installer,,mmci.dll
```

HKR stands for HKEY_ROOT, which is the base registry key for the section that the AddReg appears in. This is typically under System\Current-ControlSet\Enum\Root, then a specific key for the device.

The installer file *mmci.dll* in the example is included with Windows 98 and is stored in the *windows**system* folder.

Manufacturer

The Manufacturer section identifies the device (or devices) and names the Install section for each. Every INF file must have this section.

In the example, the MfgName string (defined later in the file) is set to the value *Lakeview*:

```
[Manufacturer]
%MfgName%=Lakeview
```

The Lakeview section has additional information:

```
[Lakeview]
;Uses Lakeview Research's Vendor ID (0925)
;Uses the Product ID 1234
%USB\VID_0925&PID_1234.DeviceDesc%=SampleHID,
USB\VID_0925&PID_1234
```

This section names the device's Vendor and Product IDs. When the Device Manager finds a match between these and the IDs retrieved from the device on enumerating, it knows that it has found the right INF file.

DestinationDirs

The DestinationDirs section names the folder or folders that any CopyFiles, RenFiles, and DelFiles items will use. In the example, SampleHID.CopyList is the name of a section that has a CopyFiles item. The value is a logical disk identifier (LDID) of 11, which is the system directory. The *Device Information (INF) File Reference* in the Windows DDK documentation lists other LDID values.

```
[DestinationDirs]
SampleHID.CopyList = 11
```

The SampleHID section has the CopyFiles item and an AddReg item:

```
[SampleHID]
CopyFiles=SampleHID.CopyList
AddReg=SampleHID.AddReg
```

These items name other sections in the file.

The SampleHID.CopyList section lists the drivers for the device:

```
[SampleHID.CopyList]
hidusb.sys
hidclass.sys
hidparse.sys
```

These are the drivers for generic HID-class devices. They're stored in *windows\system32\drivers* or *winnt\system32\drivers*.

The SampleHID.AddReg section adds registry information for the device:

```
[SampleHID.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,"hidusb.sys"
```

DevLoader names *ntkern.vxd* as the VxD (virtual driver) that loads the drivers. *Ntkern.vxd* in turn loads the driver named in NTMPDriver. In the example, this is *hidusb.sys*. Both files are included with Windows 98. You won't find the file *ntkern.vxd* on your system because it's archived in, or bound into, the file *vmm32.vxd* for quicker loading.

Strings

The Strings section defines the strings referred to by items in other sections. Each item matches an item surrounded by percent signs in another section.

So, for example, the provider in the Version section is equal to %Provider%, which equals *Microsoft* (since they are the source of the drivers).

```
[Strings]
Provider="Microsoft"
MfgName="USB Complete"
USB\VID_0925&PID_1234.DeviceDesc="Sample USB human
interface device (HID)"
```

The Generic INF File for HIDs

The generic INF file for HIDs is *hiddev.inf* in Windows 98 and *input.inf* in Windows 2000. Every Windows system should have one of these files. It's similar to the sample file in Listing 11-1. The Device Manager uses this file to install any HID that doesn't have its own INF file. The file also has Vendor and Product IDs and descriptions for several manufacturers' devices, so these don't need their own INF files.

Creating INF Files

If you need to create an INF file for a device, Microsoft provides several tools to help in creating the file and ensuring that it has all of the required sections in the correct format. This section describes the tools and also gives some tips that can come in handy when you're experimenting with INFs.

Tools

For creating INF files, Microsoft provides *infedit* for Windows 98 and *Geninf*, *ChkINF*, and *InfCatReady* for Windows 2000.

The Windows 98 DDK includes the *infedit* application (Figure 11-6), which enables you to examine and edit INF files. To protect the installed INF files, *infedit* hides the *windows\inf* folder, so to view an installed file, you'll need to copy it to a different folder. You can also use any text editor to view and edit INF files.

The Windows DDK includes two tools for Windows 2000 INF files: *Geninf* for creating files and *ChkINF* for checking a file's structure and syntax.
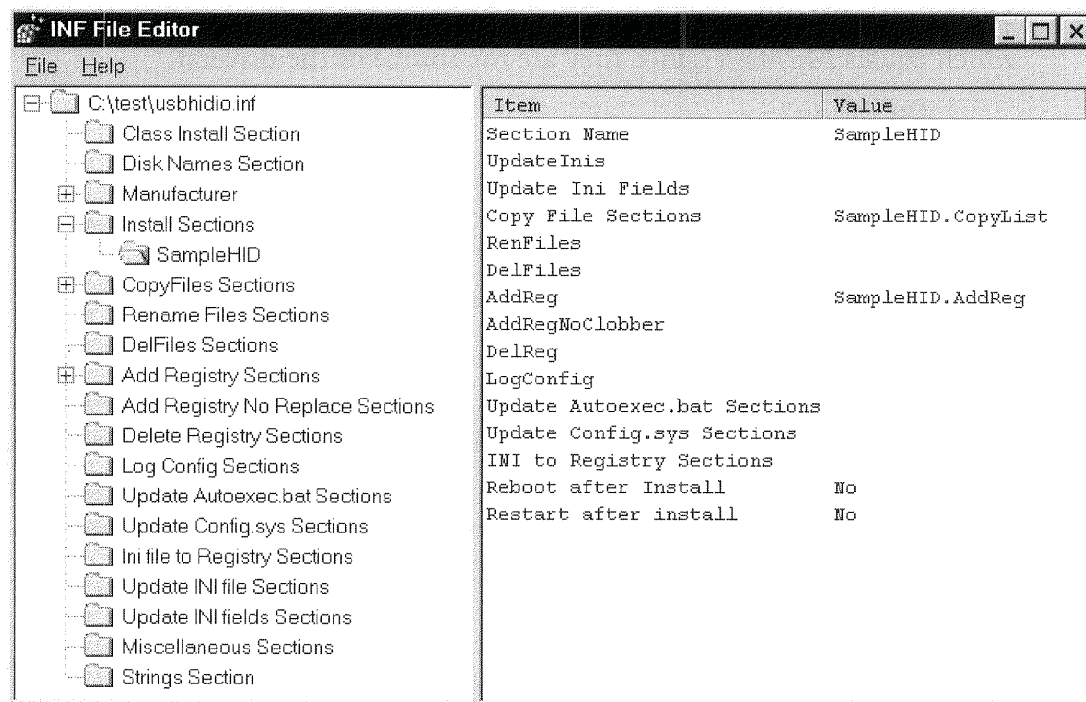
Figure 11-6: Windows 98's *infedit* tool enables you to view and edit INF files.

The *Geninf* application has an INF wizard that asks you questions about your device and creates an INF file for it. The documentation warns that the created file is a skeleton that may not be fully valid and is likely to need additions or revisions. The application includes specific support for some device classes.

ChkINF is a Perl script that requires a Perl interpreter, which you can download free from *www.activeware.com* and other sources. The script runs from an MS-DOS prompt and creates an HTML page that annotates an INF file with errors and warnings as needed.

For drivers that will use digital signing as described in Chapter 17, Microsoft provides the *InfCatReady* application, which looks for errors that could interfere with the digital signature and thus prevent driver installation. *InfCatReady* is available from the WHQL website at *www.microsoft.com/hwtest*.

Tips

Here are some tips for using and experimenting with INF files:

A commercial product's Vendor ID must be an official ID assigned by the USB Implementers Forum. My examples use the Vendor ID of 0925h, which is assigned to my company, Lakeview Research. The owner of the Vendor ID is responsible for ensuring that each product and version has a unique set of IDs. Borrowing someone else's Vendor ID can lead to conflicts if the owner of the ID uses the same values for a different product.

As described above, for experimenting with HIDs, you can use Windows' generic INF file, instead of an INF file containing your Vendor ID. The Device Manager will show the device as a generic HID, rather than using the name you provide in an INF file.

When experimenting with different settings in an INF file, you may find that at times the Device Manager remembers information from previous INF files, even if you deleted the previous file and the information about the device in the registry, powered down, and rebooted.

Under Windows 98, unless you follow a specific procedure when changing the contents of an INF file, Windows may fail to rebuild the driver information database.

To ensure that Windows 98 is aware of any changes you've made to an INF file, follow this procedure:

1. Save a copy of the new INF file that you want to use. Save it under another name (such as *mydriver.new*) or in a location other than the system's INF folder.

2. Attach the device and allow the Device Manager to enumerate it.

3. In the Device Manager's window, select the device's entry and select Remove.

4. Deleting the entry in the Device Manager causes the device's INF file to be saved in the *windows\inf\other* folder, with the vendor's name added to the beginning of the filename. For example, Lakeview's file *mydriver.inf* would become *lakeviewmydriver.inf.* Delete this file as well. In some cases,

such as the system's INF files, the *inf\other* folder won't have anything to delete.

5. Copy the INF file you want to use to the *windows\inf* folder. Be sure the file has an extension of *.inf* (such as *mydriver.inf*).

6. Unplug and re-attach the device. Windows will rebuild the driver information database using your new INF file.

Another way to accomplish the same thing under Windows 98 is described in Microsoft's article Q139206, *Hardware List Not Updated After Installing New .inf File*. The article suggests renaming the driver information database to force Windows to rebuild it. In the *windows\inf* folder, rename *drvdata.bin* to *drvdata.xxx* and rename *drvidx.bin* to *drvidx.xxx*. (By renaming the files rather than deleting them, you can restore them if necessary.) Another workaround is to use a different Product ID each time, in both the INF file and the device firmware.

Under Windows 2000, to remove all information about a device, delete or change the extension of its INF and PNF files. When Windows stores the files in *\winnt\inf*, it may rename them *oem*.inf* and *oem*pnf*, where * is a number. To find the correct files, use the *Find > Files or Folders* utility available from Windows' *Start* menu. Browse to the *\winnt\inf* folder and in the *Containing Text* text box, enter *VID_xxxx&PID_yyyy*, where *xxxx* is the vendor ID and *yyyy* is the product ID, both in hexadecimal.

If you do a lot of experimenting and don't delete each device when you're done with it, the registry will fill with entries from your various configurations. When you no longer need a registry key, you can delete it from within *regedit.exe* (but see my cautions above about the registry).

The INF files that ship with Windows all have file names with no more than eight characters plus the 3-character extension. Microsoft says that this is due to "technical issues with the product install," but that INF files added after Windows is installed may use longer file names.

12

Device Classes

Most devices aren't totally unique, but instead share many qualities with other devices. For example, all printers receive and print data and send status information back to the host. All mice send information about mouse movements and button clicks to the host. All disk drives transfer files between a disk and the host.

When a group of devices or interfaces share many attributes or when they provide or request similar services, it makes sense to define the attributes and services in a class specification. The specification then serves as a guide for device developers and device-driver writers.

This chapter describes USB's defined classes and takes a closer look at both common and more unusual peripheral types and how you can use classes to simplify developing on both the host and device sides.

Uses of Classes

Classes offer several advantages. They make it easier to develop device drivers and firmware because the work of defining the attributes and services the device will use has been done, leaving only the implementation details. If both the driver writer and firmware developer follow the same specification, the driver should have no problem communicating with the device. Windows and other operating systems include drivers for common classes. If your device's class is supported by the operating system, you don't have to provide a driver with the device.

When a device in a supported class has unique features or abilities, the device vendor can provide a filter driver that adds capabilities to the class driver included with the operating system. Adding a filter driver is easier than writing the complete driver.

Even if the device's class isn't supported by the operating system, it may be in the future. If you design the firmware and driver to comply with the class specification, it will be compatible with any driver added in future editions of the operating system.

The USB Implementers Forum releases class specifications developed by Device Working Groups whose members have expertise and interest in a particular area. A special case is the hub class, which is defined in the main USB specification rather than in its own document. The operating system must support the hub class because the host requires a root hub to do any communications at all.

Elements of a Class Specification

All class specifications are based on the Common Class specification, which describes what information a class specification should contain and how the specification document should be organized. A class specification defines the number and type of endpoints supported by the class. A specification may also define formats for data to be transferred, including both general data and status and control information. Many class specifications also define functions or capabilities that describe how the data being transferred will be

used. For example, the HID class has Usage Tables that define how to interpret data sent by keyboards, mice, joysticks, and other devices.

A class specification may define class-specific items for the standard descriptors as well as class-specific descriptors, interfaces, endpoint usages, and control requests. For example, the device descriptor for a hub includes a bDeviceClass value of 09h to indicate that the device belongs to the hub class. The hub must also have a hub-class descriptor, with a descriptor type of 29h. Hubs also support class-specific requests. When the host sends a Get_Port_Status request to a hub with a port number in the Index field, the hub responds with status information for the port. (Chapter 18 has more on hubs.) A class may also require a device to support specific endpoints or comply with tighter timing for standard requests.

Defined Classes

In addition to the hub class, specifications for several other classes have been released. However, just because a specification exists doesn't mean that Windows includes drivers for the class. Table 12-1 shows the class drivers added in each edition of Windows.

The following are classes with released specifications:

Audio Device. Devices that transfer audio, voice, or sound and related controls. Windows 98 Gold (original) and later include an audio driver. Windows 2000 and Me also have a **MIDI** driver that supports the MIDI protocol for music control.

Chip/Smart Card Interface Devices. For devices that conform to the ISO/IEC 7816 specification.

Communications Device. Telephones, modems, and other telecommunications devices. Windows 98 SE and later include a modem driver.

Content Security. Supports protected and controlled distribution of digital content.

Device Firmware Upgrade. For updating program code in a device.

Table 12-1: Microsoft adds new USB driver support with each release of Windows. The releases are listed top to bottom from earliest to latest. Each release also includes the drivers provided with earlier releases.

Windows Edition	USB Version Compliance	USB Drivers Added
Windows 98 Gold (original)	1.0	Audio
		HID 1.0 (includes keyboard, pointing devices)
Windows 98 SE	1.1	Communications (modem)
		HID 1.1 (adds the ability to do interrupt OUT transfers)
		Still image capture (scanner, camera) (first phase/preliminary)
Windows 2000, Windows Me	1.1 (2.0 support expected in an update)	Mass storage
		MIDI (in the audio driver)
		Printer. This driver can also be distributed for use with Windows 98.
		Still image capture (scanner, camera) (enhanced)

Human Interface Device (HID). Keyboards, mice, joysticks, or any device that transfers blocks of information to or from the host at moderate rates, using control or interrupt transfers. Windows 98 Gold and later include HID 1.0 drivers. Windows 98 SE and later include HID 1.1 drivers, which support interrupt OUT transfers. The **Monitor** class describes HIDs that provide user controls on display monitors (not the display interface itself). The **Physical Interface** class supports HIDs that use real-time physical feedback, such as force-feedback joysticks. The **Power** class describes HIDs that provide power-supply control, including control for power conservation and uninterruptible power supplies.

IrDA Bridge Device. To replace or supplement a motherboard-mounted IrDA transceiver.

Mass Storage. For CD-ROM, tape, floppy drives, etc. Windows 2000 and Windows Me include a mass-storage driver (*usbstor.sys*).

Printer. The printer interface (not the page-description protocols). Windows 2000 and Windows Me include a printer driver (*usbprint.sys*), and the driver can be distributed for use with Windows 98.

Imaging. For scanners and still-image (not video) cameras. Windows 98 SE included a preliminary version that was enhanced in Windows 2000 and Windows Me (*usbscan.sys*).

Other class specifications under development are Device Bay Controllers and PC Legacy Compatibility. All of the specifications are available from the USB Implementers Forum website.

For more details about a class, see the class specification and for most classes supported under Windows, the DDK has further documentation.

The provided class drivers aren't installed until a device requires them. So for example, a Windows 2000 system won't show the mass-storage driver *usb-stor.sys* until a device that requires it is attached and the device's INF file causes the driver to be installed. A driver may be archived in a file on the system's hard drive, or the user may have to insert the Windows install disk to retrieve the file.

Matching a Device to a Class

Many peripherals are standard types such as the keyboards, mice, printers, and disk drives found on most desktop systems (though not always with USB interfaces). Other peripherals perform non-standard functions such as data acquisition or motor control for specific applications. The following sections contain advice on how to select a class for various applications.

Standard Peripheral Types

Standard peripheral types are likely to have built-in drivers. For the most part, users and application programmers don't have to know or care whether a device uses USB or another interface type. The hardware-specific communications are handled at a lower level and present a common interface to applications. For example, users can access files on a hard drive in exactly the same ways whether the drive uses USB, ATAPI, SCSI, IEEE-1394, or a parallel-port interface.

Keyboard, Mouse and Joystick

The keyboard, mouse, and joystick are the big three of the HID class. "Mouse" includes trackballs and other pointing devices. HIDs also encompass various other game controls. All Windows editions support USB versions of these peripherals.

Many applications don't need to access these devices directly. For example, a Visual-Basic application doesn't have to read mouse clicks to find out if a user has clicked on an option button because the button's click event executes automatically when this occurs.

Windows provides two ways for applications to communicate directly with HIDs: Windows API functions and the APIs supported by DirectX, which enables faster, more direct access to the hardware. However, Windows 2000 doesn't allow applications to use API calls or DirectX to access the system keyboard or mouse.

Besides supporting standard peripherals, the HID class is a good, general-purpose class for other uses. For this reason, the following chapters have much more detail about how to use HIDs.

Mass Storage Devices

The mass-storage class encompasses disk drives, including floppies, hard drives, CDs, and so on. Other devices that transfer files in one or both directions can use this class as well.

On a PC, all devices that use a mass-storage driver appear as drives in My Computer. Users can use the same interface to copy, move, and delete files. For example, for a digital camera that uses a mass-storage driver, the camera's memory appears to the operating system like any other drive. There's no need for proprietary software to access the images in the camera.

The many types of media supported by the mass-storage class have different internal structures. Several industry-standard sets of command blocks, or command descriptor blocks, enable controlling and reading status information from different device types. Floppies, CDs, tape drives, and Flash memory each typically use a different command-block set.

The mass-storage class supports two transport protocols that determine which transfer types the device and host use to send command, data, and status information.

Bulk-only transport uses bulk transfers for most communications. It uses control transfers only to clear a Stall condition on a bulk endpoint and to send class-specific requests. The two class-specific requests supported are Bulk Only Mass Storage Reset (reset the device) and Get Max Lun (get the number of logical units the device supports).

Control/bulk/interrupt (CBI) transport uses bulk transfers for transferring data and control transfers to clear a Stall condition on a bulk endpoint and to send class-specific requests. The single class-specific request is Accept Device-Specific Command, which enables the host to send a command block. A CBI device may use either interrupt or control transfers to signal the completion of commands.

In the device's interface descriptor, the value 08h in the bInterfaceClass field indicates that the device is mass-storage class. The bInterfaceSubClass field specifies the supported command-block set. The bInterfaceProtocol field contains a code indicating the supported transport protocol.

There are separate specifications for each transport protocol, plus a UFI Command Specification for removable media.

There are several approaches to writing or obtaining a mass-storage driver for a device. Windows 2000 and Windows Me include a driver that supports bulk-only and CBI devices. Microsoft hasn't provided much documentation for the driver, but the class specification can serve as a guide to firmware design, and applications can access devices in the same way they access other system drives.

Windows 98 doesn't have a mass-storage driver, so device vendors will have to provide one. Microsoft provides source code for a mass-storage driver for use under Windows 98 (described in knowledge base Article ID Q257751). Cypress Semiconductor has a mass-storage reference design for its EZ-USB chip. The design works with Windows 2000's driver and with a free driver provided by Cypress for use with Windows 98.

For OEMs (original equipment manufacturers) whose existing devices have standard SCSI, ATA, or ATAPI interfaces, SCM Microsystems has USB Intelligent Cables and drivers that quickly add USB capability to the devices. Many hard drives, CD drives, tape drives, and some scanners use either SCSI, ATA (AT attachment), originally known as IDE, or ATAPI (AT attachment packet interface), an extension to EIDE. The EUSB-S1 product contains a microcontroller and an ASIC that convert between the device's existing SCSI interface and USB. In a similar way, the EUSB-C product converts between ATA and ATAPI devices and USB. The cables are available only to OEMs, not to end users.

Printers

Windows 2000 and Windows Me include a USB printer driver and Microsoft also permits distributing the driver for use with Windows 98. The printer vendor must supply a high-level, user-mode driver that is layered above the print spooler. The interface to the USB printer driver is similar to the interface for parallel printers, so a single driver often works without modification with both USB and the parallel port.

Cameras and Scanners

The still-image capture, or imaging, specification was created to support still-image (not video) digital cameras. Other devices that have similar requirements, such as scanners, fit into the class as well. Version 1.0 was released in July 2000.

The Photographic and Imaging Manufacturers Association (PIMA) developed the PIMA 15740 Standard, which describes requirements for transferring files and for controlling digital still cameras. USB's specification is based on this standard.

The class supports bulk IN and bulk OUT endpoints for sending both image and non-image data, plus an interrupt IN endpoint for event data. Three class-specific requests are required and one is optional. The required requests are Cancel Request (cancel a bulk transfer), Device Reset Request (the device returns to the Idle state if the bulk pipe has stalled), and Get Device Status (the host receives information about a transfer cancelled by

the device). Optional is Get Extended Event Data (the device returns information about an event or condition.)

The interface descriptor in the device identifies a still-image device, with the bInterfaceClass field set to 06h to indicate an image interface and bInterfaceSubClass set to 01h to indicate a still-image capture device.

Windows 98 SE included a preliminary version of a still-image driver that was enhanced in Windows 2000 and Windows Me. The driver supports USB, SCSI, and IEEE-1394.

Windows 2000 and Windows Me support the Microsoft Windows Image Acquisition (WIA) architecture, which is built on the Microsoft Still Image Architecture (STI) used in previous Windows editions. The device vendor needs to supply only a user-mode WIA minidriver that provides a device-specific interface to the generic still-image driver. The Windows DDK has more details about how to use the driver.

For Windows 98 Gold and probably Windows SE, you'll need to provide a device driver.

If all that is needed is a way to transfer image files from a camera, another option is to use a mass-storage driver, as described earlier.

Audio Applications

Audio has been supported beginning with Windows 98 Gold, so there should be no need to write an audio driver. Windows 2000 and Windows Me added a MIDI driver. Audio functions are often part of a device that also supports video, storage, or other functions.

An audio function consists of an Audio Interface Collection containing one or more device interfaces. The AudioControl interface accesses controls such as volume, mute, bass, and treble. One or more AudioStreaming interfaces transport data representing audio to or from the device. One or more MIDIStreaming interfaces transport MIDI data to or from the device.

The default control endpoint responds to class-specific requests. Isochronous endpoints transfer data for the streaming interfaces. Some isochronous

endpoints may require an additional isochronous synch endpoint. An optional interrupt IN endpoint transfers status information.

MIDI (musical instrument digital interface) is a standard for controlling synthesizers, sound cards, and other electronic devices that generate music. A MIDI representation of a sound includes values for pitch, length, volume, and other characteristics. A pure MIDI hardware interface carries asynchronous data at 31.25 kilobits per second. USB MIDI carries MIDI data but doesn't use MIDI's hardware interface.

The audio and MIDI specifications have the details needed to implementing an audio interface.

Modems

The modem driver included with Windows 98 SE and later (*usbser.sys*) is compatible with modems that use the Abstract Control Model defined in the communications class specification. A modem used by programs that call the Windows Telephony Application Programming Interface (TAPI) to make data, fax, or voice calls must have its own INF file; descriptors that place the device in the communications class aren't sufficient. The Windows DDK includes a Modem Development Kit with tools, sample INF files, and information for creating and testing INF files for AT (data) and AT+V (data + voice) command modems.

Non-standard Functions

One of the great things about USB is that you're not limited to a few standard peripheral types. Applications can communicate with any peripheral if the operating system has a driver for the it. Some peripherals require custom drivers. But even when a device's purpose is very different from typical peripherals, it's often possible to design the device to fit into a defined class.

Devices that Transfer Data at Moderate Speeds

Motor controllers and data-acquisition units are two examples of specialized peripherals that aren't found on most PCs. For a motor controller, the host may send configuration and control requests to the device, which then pro-

vides the signals required to carry out the requested tasks. A controller may also send status information to the host. For data acquisition, a device may collect data from sensors and sends the results periodically to the host, and the host may send configuration or control requests to the device.

For devices in both of these categories, or any device that transfers data at low to moderate speeds, you may be able to design the device to fit the HID class, eliminating the need to provide a custom driver.

A HID doesn't have to be a standard peripheral type, and it doesn't even need a human interface. The only requirement is that the descriptors stored in the device must conform to the requirements for 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 later, including Windows 2000 and Me, will use interrupt transfers if an OUT interrupt pipe is available. Otherwise the host will use control transfers to send data to the device. The original release of Windows 98 complies only with the HID 1.0 specification and uses control transfers for all host-to-device data.

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 the class attractive when the limits are acceptable.

Upgrading RS-232 Devices

The RS-232 serial port is a good, general-purpose interface that has been with the PC since its beginning. There are probably thousands of different RS-232 peripherals in use. Microsoft and Intel's *PC 2001 System Design Guide* doesn't forbid RS-232 ports, but it discourages them in favor of newer, more powerful and flexible interfaces like USB. Just about any device

that uses RS-232 can be implemented with USB. There are several approaches to making the switch.

RS-232 modems of course can be designed for USB's modem class.

For many other devices, FTDI's FT8U232AM USB UART provides a quick way to upgrade a design to USB. The chip converts an existing RS-232 serial device to USB while requiring minimal design changes and no changes to host software. (Figure 12-1).

A typical device with an RS-232 interface contains a UART that converts between the serial data used in RS-232 communications and the parallel data used by the CPU's internal buses. The signals on the line side of the UART connect to converters that translate between RS-232 voltages and the 5V logic used by the CPU. The line side of the converter connects to a cable that connects to the remote device.

The USB UART converts between USB and RS-232, including not just the data lines but also RTS, CTS, and the other status and control signals used in RS-232 communications. One set of pins on the USB UART looks like the line side of a conventional UART, with pins for data and handshaking signals. Two other pins connect to a USB transceiver.

The chip requires no programming except the optional storing of Vendor, Product, and Device IDs and strings in a serial EEPROM.

To adapt an RS-232 design for USB, you replace the original UART's connections to the RS-232 converters with connections to the complimentary signals on the USB UART. Store the IDs and other optional information in a serial EEPROM that connects to the USB UART and add a USB connection to the USB transceiver. The device firmware requires no changes because the original UART will think it's talking to an RS-232 device as usual.

But providing the device hardware is only half of the job. The other half is the device driver. For the least disruption to existing applications, the driver should cause application software to treat the device as if it were still attached to a COM port. FTDI provides drivers that do just that under
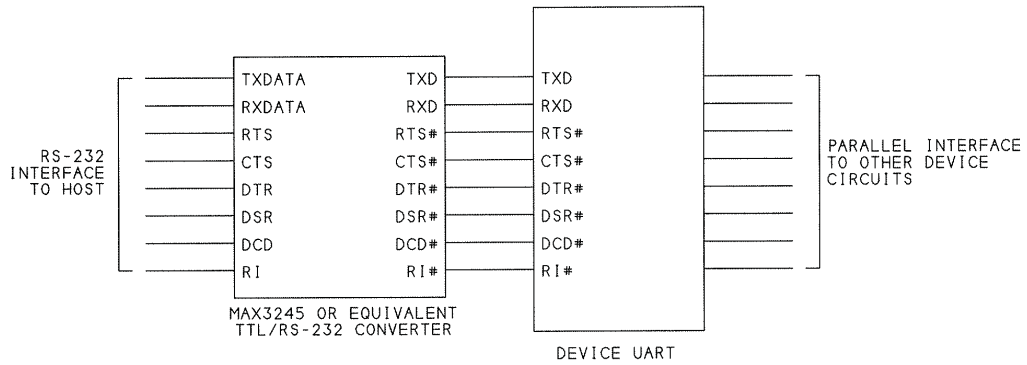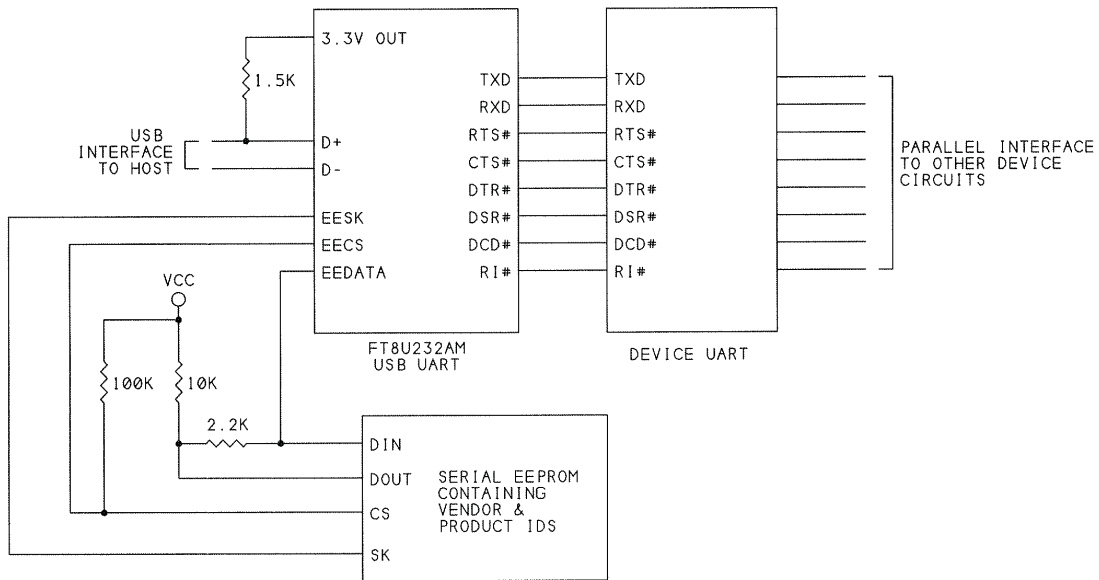
Figure 12-1: FTDI's USB UART can convert devices with RS-232 interfaces to USB. A free device driver provided by FTDI causes the device to appear like a conventional COM-port device to host applications.

Windows and other operating systems. An RS-232 design converted for USB with an FTDI UART can use exactly the same application software as the RS-232 version.

Another approach to upgrading RS-232 devices is to redesign the device to eliminate the COM-port interface entirely. The device will probably be cheaper to manufacture because there's no need for a UART, but the device

will need new application software and possibly a custom device driver. Many RS-232 devices, such as uninterruptible power supplies and the point-of-sale devices described below, can be designed as HIDs. Others will use bulk transfers and may require a custom driver.

Point-of-Sale Devices

Point-of-sale (POS) devices include bar-code scanners, displays, receipt printers, cash drawers, coin dispensers, and other devices used in sales transactions. Traditionally these have used RS-232 interfaces, and they're ideal candidates to upgrade to USB.

Most POS devices can be designed to fit into the HID class. The *HID Point of Sale Usage Tables* document defines data formats for bar-code scanners, weighing devices, and magnetic stripe readers. The document is available from the USB Implementers Forum's website.

Other approaches for POS devices are designed to make upgrading from RS-232 as easy as possible. RS-232 POS devices can use the USB UART described above to enable applications to access the device the same as if it were still connected to a COM port.

Another option is the EPiC driver and associated USB protocol from Inside/Out Networks. The driver enables applications to access a device as if it were a COM-port device. This approach requires the device to contain a USB controller with device firmware that uses the licensed protocol.

Replacing Non-standard Parallel Port Devices

Besides the RS-232 serial port, another port that all PCs had from the beginning was the parallel port, originally intended for connecting a printer. Like the serial port, the parallel port has found many other uses over the years. The parallel port is faster than the serial port, so it became a favored connection for scanners and disk drives. This became even more true when the ports began supporting the new, faster PS/2, enhanced parallel port (EPP) and extended capabilities port (ECP) modes. In each of its modes, the parallel port uses a defined protocol for exchanging bytes of data along with status and control information.

Another category of parallel-port devices uses custom protocols. The original port had 8 outputs, 5 inputs, and 4 open-collector, bidirectional lines. Under Windows 3.x and 9x, applications can read and write directly to the port addresses, and under Windows NT and 2000 all that's needed to access the ports is a kernel-mode driver available at low cost or free from several sources. What resulted was an assortment of devices following no standard use of the port's input and outputs. For example, one popular use involved connecting combinations of decoders, flip-flops, and data selectors to expand the number of inputs and outputs applications could access.

But as with RS-232, Microsoft and Intel are discouraging the parallel port's use in favor of USB and IEEE-1394. And this brings up the question of what to do with all of the existing designs.

For drives, scanners, and other standard device types, the logical solution is to design the device to comply with the appropriate USB class specification.

A quick solution for parallel printers is to use a USB printer adapter. The adapter's driver causes the operating system to see the printer as a network printer. Adapters are available from several vendors. A printer adapter isn't a solution for parallel-port scanners, drives, and so on, because the firmware and driver are designed for use only with the PC's printer drivers.

For devices that use non-standard parallel-port communications, the solution is to redesign the interface for USB. This requires adding a USB micro-controller to the device, possibly providing a device driver, and revising the application software to match the driver's requirements. The parallel port has 17 signal pins, so to emulate them all requires at least that many I/O pins on the microcontroller. But many designs can get by with the 16 I/O pins available on smaller, cheaper controller chips. If you must have 17 bits on a chip with a small footprint, Cypress' CY7C63743 has 16 I/O pins plus two additional inputs that are available if the chip uses the internal oscillator or an external source for D-'s pull-up.

Applications that access the port at low and moderate speeds can probably use the HID drivers included with Windows. This means there are no drivers to write, but you'll need to rewrite the application software to use the API calls for accessing HIDs.

If you want to make minimal changes to the application software, provide a driver that supports custom DeviceIoControl functions that emulate the functions used by the original application. For example, you could define an IoControlCode for a status-port read function that reads five inputs with bit values of bit 3 through bit 7 and even inverts bit 7 to match what the parallel-port hardware does. Instead of reading the status-port address with an Inp function, applications would call DeviceIoControl with your IoControlCode for the status-port read emulation.

PC-to-PC Connections

USB doesn't allow peripherals to exchange data directly. All communications must go through a host. There's no way for two hosts to send data to each other without going through a peripheral. There is, however, a way to enable two PCs to communicate using their USB ports. Each PC can connect to a USB peripheral, and the two peripherals can communicate with each other via a shared buffer.

Cypress Semiconductor's AN2720SC is designed for this purpose. It's a single chip containing two USB cores. Each core connects to a USB transceiver and a shared 2-Kilobyte buffer. Cypress provides a driver that causes each PC to see the other as a network-connected PC. You add only a single crystal, an EEPROM for storing a VID and PID, and few other components.

But you don't have to build your own PC-to-PC cable. It's a popular enough application that ready-made products are available, including Cypress' EZ-Link.

Wireless Links

Replacing a USB cable with a wireless connection isn't a simple task. The main reason is that USB transactions involve communicating in both directions with tight timing requirements. For example, when a host sends a token and data packet in the data stage of an interrupt transaction, the device must respond quickly with ACK or another code in the handshake packet. Designing a wireless link to do this while also meeting all of USB's timing and other requirements would be a challenge.

An easier solution when you need a wireless connection is to use a conventional wired connection to a USB device that also supports a wireless interface. The device at the other end of the wireless link doesn't have to support USB at all.

SigmaTel's STIr4200s takes this approach with its IrDA-to-USB bridge chip for wireless applications. IrDA is a standard for communications that use infrared energy instead of cables. The bridge's USB interface connects to a USB hub, and the IrDA interface communicates with IrDA-capable devices. The bridge translates between the two interfaces. SigmaTel provides a driver for use with the chip.

A similar approach would work for devices that use radio-frequency wireless communications.

13

Human Interface Devices: Firmware Basics

The human interface device (HID) class was one of the first USB classes to be supported under Windows. On PCs running Windows 98 or later, applications can communicate with HIDs using the drivers built into the operating system. For this reason, USB devices that fit into the HID class are some of the easiest to get up and running.

This chapter shows how to determine whether a peripheral will fit into the human-interface class, explains the firmware requirements that define a device as a HID and enable it to exchange data with its host, and introduces the six HID-specific control requests. The next three chapters describe the reports that HIDs use to exchange information and how to access HIDs from applications.

What is a HID?

Before you can know whether or not you can use Windows' HID drivers to communicate with a device, you need to know whether your device fits in the HID class.

The designation *human interface* suggests that the device interacts directly with people. A device may detect when someone presses a key or moves a mouse or joystick, or the host may send a message that translates to a joystick effect that the user experiences. The classic examples of HIDs are keyboards, mice, and joysticks. Other HIDs include front panels with knobs, switches, buttons, and sliders; remote controls; telephone keypads; and game controls such as data gloves and steering wheels.

But a HID doesn't have to have a human interface at all. It just needs to be able to function within the limits of the class's specification. These are the major abilities and limitations of HID-class devices:

- The data exchanged resides in structures called reports. The device's firmware must support the HID report format. The host sends and receives data by sending and requesting reports in control or interrupt transfers. The report format is flexible, and can handle just about any type of data.

- Each transaction can carry a small to moderate amount of data. For a low-speed device, the maximum is 8 bytes per transaction. For a full-speed device, the maximum is 64 bytes per transaction. For a high-speed device, the maximum is 1024 bytes per transaction. A long report can use multiple transactions.

- A device may send information to the computer at unpredictable times. For example, there's no way for the computer to know when the user will press a key on the keyboard, so the host's driver polls the device periodically to obtain new data.

- The maximum speed of transfers is limited, especially at low and full speeds. As Chapter 4 explained, a host can guarantee a low-speed interrupt endpoint no more than 1 transaction per 10 milliseconds, for a maximum of 800 bytes per second. A host can guarantee a full-speed endpoint up to 1 transaction per millisecond, for a maximum of 64,000

bytes per second, or a high-speed endpoint up to 3 transactions per 125 microseconds, for a maximum of 24.576 Megabytes per second.

- There is no guaranteed *rate* of transfer. If the device is configured for 10-millisecond intervals, the time between transactions may be any period equal to or less than this. The exception is devices configured to transfer data every frame at full speed or every microframe at high speed. Since these are the fastest possible polling rates, the endpoint is guaranteed to have this exact bandwidth available.

- Under Windows 98 Gold (original), interrupt OUT transfers aren't supported, so all host-to-device data must use control transfers.

Although many HIDs mostly send data from the device to the host, a HID can also receive data from the host. The classic example of host-to-device HID communications is the force-feedback joystick, where users experience effects that match their actions, such as greater resistance when pulling the stick to cause a simulated airplane to climb or when getting a bite on a simulated fishing rod.

Any device that can live within the class's limits is a candidate to be a HID. The specification mentions bar-code readers, thermometers, and voltmeters as examples of HIDs that may not have a conventional human interface. Each of these sends data to the computer and may also receive requests that configure the device. Examples of devices that mostly receive data are remote displays, control panels for remote devices, robots, and devices of any kind that receive occasional or periodic commands from the host.

The HID interface may be just one of multiple USB interfaces supported by a device. A video display may have a HID interface for software control of brightness, contrast, and refresh rates, while using the conventional video interface to send the data to be displayed. A USB speaker that uses isochronous transfers for audio may also have a HID interface for controlling volume, balance, treble, and bass. A HID interface is often cheaper than traditional physical controls.

Two essential documents for working with HIDs are *Device Class Definition for Human Interface Devices*, which defines the HID class, and *HID Usage Tables*, which defines values that help the host understand and use the HID

data. Both documents are products of a USB Device Working Group. The members are affiliated with the member companies of the USB Implementers Forum. The documents are published by the Implementers Forum and available on the Forum's website.

Hardware Requirements

A HID interface must conform to the requirements of the HID class as defined in the specification. The document describes the required descriptors, the frequency of transfers, and the transfer types available.

To comply with the specification, the interface's endpoints and descriptors must meet several requirements.

Endpoints

All HID transfers use either the Default Control Pipe or an interrupt pipe. A HID must have an interrupt IN endpoint for sending data to the host. An interrupt OUT endpoint is optional.

The specification defines uses for each pipe. Table 13-1 shows the transfer types and their uses in HIDs.

You can think of the data that the host and device exchange as being of two types: low-latency data that must get to its destination as soon as possible, and configuration data or other data that doesn't have critical timing requirements. (By configuration data, I'm referring to data sent in HID reports, not the host's requesting and selecting of device configurations on enumerating.)

The Control Pipe

The control pipe for a HID carries the standard USB requests as well as six class-specific requests defined in the HID specification. Two of the HID-specific requests, Set_Report and Get_Report, provide a way for the host and device to transfer a block of any kind of data to or from the device. The host uses Set_Report to send reports and Get_Report to receive reports.

The other four requests relate to configuring the device. Set_Idle and Get_Idle set and read the Idle rate, which determines whether or not a

Table 13-1: The transfer type used in a HID transfer depends on the chip's abilities and the requirements of the data being sent.

Transfer Type	Source of Data	Type of Data	Required Pipe?	Windows Support
Control	Device (IN transfer)	Data that doesn't have critical timing requirements.	yes	Windows 98 and later
	Host (OUT transfer)	Data that doesn't have critical timing requirements, or any data if there is no OUT interrupt pipe.		
Interrupt	Device (IN transfer)	Periodic or low-latency data.	yes	
	Host (OUT transfer)	Periodic or low-latency data.	no	Windows 98 SE and later

device resends data that hasn't changed since the last poll. Set_Protocol and Get_Protocol set and read a protocol value, which can enable a device to function with a simplified protocol when the HID drivers aren't loaded on the host.

Interrupt Transfers

The interrupt pipe or pipes provide an alternate way of exchanging device data, especially when the receiver must get the data quickly or periodically. An interrupt IN pipe carries data to the host, and an interrupt OUT pipe carries data to the device. Control transfers can be delayed if the bus is very busy, but once the device is configured, the bandwidth for interrupt transfers is guaranteed to be available. HIDs aren't required to have interrupt OUT pipes. If there is no interrupt OUT pipe, the host sends all reports on the control pipe, using Set_Report requests.

The ability to do Interrupt OUT transfers was added in version 1.1 of the USB specification, and the option to use an interrupt OUT pipe was added to version 1.1 of the HID specification. A HID driver that complies only with version 1.0 (including the drivers in Windows 98 Gold) won't support interrupt OUT transfers.

Firmware Requirements

For the host's drivers to communicate with a HID, the device's firmware must meet certain requirements. The device's descriptors must identify the device as having a HID interface, and the firmware must support an interrupt IN endpoint in addition to the Default Control Pipe. The firmware must also contain a report descriptor that defines the format for transmitted and received device data.

To send data, the specification requires the firmware to support Get_Report control transfers and interrupt IN transfers, and to receive data, the firmware must support Set_Report control transfers and may also support interrupt OUT transfers.

All HID data must use a defined report format that defines the size and contents of the data in the report. Devices may support one or more reports. A report descriptor in the device's firmware describes the reports, and may also include information about how the receiver of the data should use it.

A value in each report defines the report as an Input, Output, or Feature report. The host receives data in Input reports and sends data in Output reports. Feature reports may travel in either direction.

For Input reports, the HID drivers in all releases of Windows 98 and later use interrupt transfers. For Output reports, the transfer type depends on what endpoints the device supports and which edition of Windows is installed. The original release of Windows 98 (Windows 98 Gold) complies only with version 1.0 of the HID specification, and the HID driver uses control transfers for Output reports. Windows 98 SE, Windows 2000, and Windows Me comply with version 1.1 of the specification, so the HID driver uses interrupt transfers for Output reports if the interface has an interrupt OUT endpoint. Otherwise it uses control transfers. If the HID interface doesn't have an interrupt OUT endpoint or if the firmware supports both transfer types for Output reports, the HID will be compatible with any Windows edition. Feature reports always use control transfers.

A report format can be simple or complex. The rest of this chapter and Chapter 14 have much more about report formats.

Identifying a Device as a HID

As with any USB device, a HID's descriptors tell the host what it needs to know to communicate with the device. Listing 13-1 shows example device, configuration, interface, class, and endpoint descriptors for a HID-class joystick. The host learns about the HID interface when it sends a Get_Descriptor request for the configuration containing the HID interface. The configuration's interface descriptor identifies the interface as HID-class. The HID class descriptor specifies the number of report descriptors supported by the interface. During enumeration, the HID driver retrieves the HID class and report descriptors.

Descriptor Contents

The device and configuration descriptors have no HID-specific information. The device descriptor contains a field for a class code, but this isn't where the device is defined as a HID. Instead, the interface descriptor is where the host learns that a device, or more properly, a device interface, belongs to the HID class. If the class-code byte in the device's interface descriptor is 3, the interface is a HID.

Other fields that contain HID-specific information in the interface descriptor are the subclass and protocol fields, which can specify a boot interface.

Boot Interfaces

The subclass field has just one active setting. A subclass of 1 indicates that the device supports a boot interface. When a device has a boot interface, the device will be usable when the host's HID drivers aren't loaded. This might occur when the computer boots directly to DOS, or when viewing the system setup screens that you can access on bootup, or when using Windows' Safe mode for system troubleshooting. A keyboard or mouse with a boot interface can use a predefined, simplified protocol supported by the BIOS of many hosts. The BIOS loads from ROM or other non-volatile memory on bootup and is available in any operating-system mode. The HID specification defines boot-interface protocols for keyboards and mice.

```
device_desc_table:
    db 12h              ; Descriptor length (18 bytes)
    db 01h              ; Descriptor type (Device)
    db 00h,01h          ; Complies to USB Spec. Release (1.00)
    db 00h              ; Class code (0)
    db 00h              ; Subclass code (0)
    db 00h              ; Protocol (No specific protocol)
    db 08h              ; Max. packet size for EP0 (8 bytes)
    db B4h,04h          ; Vendor ID (Cypress)
    db 1Fh,0Fh          ; Product ID (joystick = 0x0F1F)
    db 88h,02h          ; Device release number (2.88)
    db 00h              ; Mfr string descriptor index (None)
    db 00h              ; Product string descriptor index (None)
    db 00h              ; Serial No. string descriptor index (None)
    db 01h              ; Number of possible configurations (1)
end_device_desc_table:

config_desc_table:
    db 09h              ; Descriptor length (9 bytes)
    db 02h              ; Descriptor type (Configuration)
    db 22h,00h          ; Total data length (34 bytes)
    db 01h              ; Interface supported (1)
    db 01h              ; Configuration value (1)
    db 00h              ; Index of string descriptor (None)
    db 80h              ; Configuration (Bus powered)
    db 32h              ; Maximum power consumption (100mA)

Interface_Descriptor:
    db 09h              ; Descriptor length (9 bytes)
    db 04h              ; Descriptor type (Interface)
    db 00h              ; Number of interface (0)
    db 00h              ; Alternate setting (0)
    db 01h              ; Number of endpoints supported
    db 03h              ; Class code (HID)
    db 00h              ; Subclass code (None)
    db 00h              ; Protocol code (None)
    db 00h              ; Index of string(None)
```

Listing 13-1: Descriptors for a HID-class joystick (Sheet 1 of 2)

```
Class_Descriptor:
      db 09h              ; Descriptor length (9 bytes)
      db 21h              ; Descriptor type (HID)
      db 00h,01h          ; HID class release number (1.00)
      db 00h              ; Localized country code (None)
      db 01h              ; # of HID class descriptors to follow (1)
      db 22h              ; Report descriptor type (HID)
                          ; Total length of report descriptor
      db (end_hid_report_desc_table - hid_report_desc_table),00h

Endpoint_Descriptor:
      db 07h              ; Descriptor length (7 bytes)
      db 05h              ; Descriptor type (Endpoint)
      db 81h              ; Encoded address (Respond to IN, 1 endpnt)
      db 03h              ; Endpoint attribute (Interrupt transfer)
      db 06h,00h          ; Maximum packet size (6 bytes)
      db 0Ah              ; Polling interval (10 ms)

end_config_desc_table:
```

Listing 13-1: Descriptors for a HID-class joystick (Sheet 2 of 2)

If a device does have a boot interface, the protocol field indicates if the device supports the keyboard (1) or mouse (2) interface. A value of zero indicates no device, and values 3–255 are reserved. A subclass of zero means that the device doesn't support a boot protocol. Values 2 through 255 are reserved.

The HID Usage Tables document defines the keyboard and mouse boot descriptors. The BIOS doesn't need to read a descriptor from the device because it knows what the boot protocol is and assumes that the device will support it. So a boot device doesn't have to include a boot-interface descriptor in firmware; it just has to support the boot protocol if the host hasn't requested the protocol defined in the report descriptor. When the operating system loads, the HID drivers use the HID-specific request Set_Protocol to cause the device to switch from the boot protocol to the report protocol.

Draft 4 Compliance

During the development of the HID 1.0 specification, a change was made to the ordering of descriptors in HID firmware. In the early versions, the descriptors were stored and retrieved in this order:

> Configuration
> Interface
> Endpoint
> HID

By Draft 4 of the specification, the order had changed to:

> Configuration
> Interface
> HID
> Endpoint

The change means that the HID descriptor is associated with an interface, rather than an endpoint. If a HID has two endpoints, the device doesn't need a HID descriptor for each.

A device that complies with HID 1.0 or later uses the Draft 4 ordering. A USB test utility (such as HIDView, described in Chapter 17) that checks for Draft 4 compliance is examining the order of the descriptors.

HID Class Descriptor

The main purpose of the HID class descriptor is to identify additional descriptors for use in HID communications. The class descriptor has seven or more fields, depending on the number of additional descriptors. Table 13-2 shows the fields.

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The value 21h indicates the HID class.

Table 13-2: The HID class descriptor has 7 or more fields in 9 or more bytes.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	21h indicates the HID class
2	bcdHID	2	HID specification release number (BCD)
4	bCountryCode	1	Numeric expression identifying the country for localized hardware (BCD)
5	bNumDescriptors	1	Number of subordinate class descriptors supported
6	bDescriptorType	1	The type of class descriptor
7	wDescriptorLength	2	Total length of report descriptor
9	bDescriptorType	1	Constant identifying the type of descriptor. Optional, for devices with more than one descriptor.
10	wDescriptorLength	2	Total length of descriptor. Optional, for devices with more than one descriptor. May be followed by additional wDescriptorType and wDescriptorLength fields.

The Class

bcdHID. The HID specification number that the device and its descriptors comply with. In BCD (binary-coded decimal) format. The value is a 4-character hexadecimal value with a decimal point assumed in the middle. For example, Version 1.0 is 0100h; Version 1.1 is 0110h.

bCountryCode. If the hardware is localized for a specific country, this field is a code identifying the country. The HID specification lists the codes. If the hardware isn't localized, this field is 00h.

bNumDescriptors. The number of class descriptors that are subordinate to this descriptor.

bDescriptorType. The type (report or physical) of a descriptor that is subordinate to the HID class descriptor. Every HID must support at least one report descriptor. An interface may support multiple report descriptors and one or more physical descriptors.

wDescriptorLength. The length of the descriptor described in the previous field.

Additional bDescriptorType, wDescriptorLength (optional). If there are additional subordinate descriptors, the descriptor type and length for each follow in sequence.

Report Descriptors

A report descriptor defines the format and uses of the data that carries out the purpose of the device. If the device is a mouse, the data reports mouse movements and button clicks. If the device is a relay controller, the data contains codes that specify which relays to open and close.

The report descriptor needs to be flexible enough to handle devices with very different purposes. The data should be stored in a concise form so it doesn't waste storage space in the device or bus time when the data transmits. The HID report descriptor achieves both of these at a price of a format that's more complex and less readable than a more verbose format might be. The format doesn't limit the type of data in a report, but the report descriptor must describe the size and contents of the report in advance. A report descriptor's contents and length vary with the device, and can be short and simple, long and complex, or anywhere in between.

A report descriptor is a type of class descriptor. The host retrieves the descriptor by sending a Get_Descriptor request with the Value field containing 22h in the high byte and the report ID in the low byte. The default report ID is 00h.

One way to get a feel for what a report descriptor contains and how it's structured is to look at one. Listing 13-2 is a bare-bones report descriptor that describes an Input report that sends two bytes of data to the host and an Output report that sends two bytes of data to the device. Other report descriptors build on this basic format, so a short descriptor like this is a good place to start understanding report descriptors in general.

The items in the example descriptor are required in all descriptors. Some items apply to the entire descriptor, while others are specified separately for the input and output data. More complicated report descriptors may use additional instances of these same items along with other optional items.

```
hid_report_desc_table:

        db 06h, A0h, FFh ;          Usage Page (vendor defined)
        db 09h, A5h      ;          Usage (vendor defined)

        db A1h, 01h      ;          Collection (Application)
        db 09h, A6h      ;          Usage (vendor defined)

;The input report
        db 09h, A7h      ;          Usage (vendor defined)
        db 15h, 80h      ;          Logical Minimum (-127)
        db 25h, 7Fh      ;          Logical Maximum (128)
        db 75h, 08h      ;          Report Size (8)   (bits)
        db 95h, 02h      ;          Report Count (2)   (fields)
        db 81h, 02h      ;          Input (Data, Variable, Absolute)

;The output report
        db 09h, A9h      ;          Usage (vendor defined)
        db 15h, 80h      ;          Logical Minimum (-128)
        db 25h, 7Fh      ;          Logical Maximum (127)
        db 75h, 08h      ;          Report Size (8)   (bits)
        db 95h, 02h      ;          Report Count (2)   (fields)
        db 91h, 02h      ;          Output (Data, Variable, Absolute)

        db C0h           ;          End Collection

end_hid_report_desc_table:
```

Listing 13-2: This report descriptor enables sending and receiving of two bytes.

Each item in the example report consists of a byte that identifies the item and one or more bytes containing the item's data. Here is what each item in the example descriptor specifies:

The **Usage Page** item is identified by the value 06h and specifies the general function of the device, such as generic desktop control, game control, or alphanumeric display (to name just a few). You can think of the Usage Page as a subset of the HID class. In the example descriptor, the Usage Page is the vendor-defined value FFA0hh. The HID specification lists values for different Usage Pages and values reserved for vendor-defined Usage Pages.

The **Usage** item is identified by the value 09h and specifies the function of the individual report. Just as the Usage Page is a subset of the class, the Usage is a subset of the Usage Page. For example, Usages available for generic desktop controls include mouse, joystick, and keyboard. Because the example's Usage Page is vendor-defined, all of the Usages in the Usage Page are vendor-defined also. In the example, the Usage is A5h.

The **Collection (Application)** item begins a group of items that together perform a single function, such as keyboard or mouse. Each report descriptor must have an Application Collection to enable Windows to enumerate it. The Usage item that follows the Collection item names the function of the collection. In the example, it's the vendor-defined value A6.

The Logical Minimum and Maximum have values of 15h and 25h and specify the range of values that the report can contain. Negative values may be expressed as two's complements. In the example, the values 80h and 7Fh indicates a range of -128 to +127.

The Report Size item has a value of 75h and indicates how many bits are in each reported data item. In the example, each data item is eight bits.

The Report Count item has a value of 95h and indicates how many data items the report contains. In the example, each report contains two data items.

The final item specifies whether the report carries data from the host to the device (91h) or from the device to the host (81h), along with other information about the data.

The End Collection item closes the Application Collection.

HID-specific Requests

The HID specification defines six HID-specific control requests. Table 13-3 lists the requests, and the following pages describe each request in more detail.

All HIDs must support Get_Report, and boot devices must support Get_Protocal and Set_Protocol. The other requests (Set_Report, Get_Idle,

Table 13-3: In addition to the eleven standard control requests, HIDs may support up to six HID-specific requests.

Request #	Request	Data source	Value	Index	Data Length (bytes)	Data stage contents	Required?
01h	Get_ Report	device	report type, report ID	interface	report length	report	yes
02h	Get_ Idle	device	report ID	interface	1	idle duration	no
03h	Get_ Protocol	device	0	interface	1	protocol	required for boot devices
09h	Set_ Report	host	report type, report ID	interface	report length	report	no
0Ah	Set_ Idle	host	idle duration, report ID	interface	0	none	no
0Bh	Set_ Protocol	host	protocol	interface	0	none	required for boot devices

and Set_Idle) are optional. If a device doesn't have an Interrupt OUT endpoint or if it is communicating with a 1.0 host such as Windows 98 Gold, it will need to support Set_Report to receive data from the host. Devices that don't support Feature reports will send data using interrupt transfers only and thus have no use for Get_Report, but to comply with the specification, they should support the request in case a host should decide to use it. A device will enumerate and transfer data under Windows without supporting this request, however.

Get_Report

Purpose: Enables the host to receive data from a device in control transfers.

Request Number: 01h

Source of Data: device

Data Length: length of the report

Contents of Value field: The high byte contains the report type (1=Input, 2=Output, 3=Feature), and the low byte contains the report ID. The default report ID is 0.

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: the report

Comments: The HID specification advises that the host should not use this request to obtain periodic data. (It should use interrupt transfers instead.) The request is intended only for obtaining the state of feature items or other information that the host needs to know when it initializes the device. However, a host using a boot protocol might use Get_Report to receive keypress or mouse data.

All HIDs must support this request.

Set_Report

Purpose: Enables a device to receive data from the host in control transfers.

Request Number: 09h

Source of Data: host

Data Length: length of the report

Contents of Value field: The high byte contains the report type (1=Input, 2=Output, 3=Feature), and the low byte contains the report ID. The default report ID is 0.

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: the report

Comments: If a device doesn't have an Interrupt OUT endpoint or if the host complies only with version 1.0 of the HID specification, this request is the only way the host can send data to the device. For other devices, the host may use this request to send Feature reports or other information that that isn't time-sensitive. HIDs aren't required to support this request.

Get_Idle

Purpose: The host reads the current Idle rate from a device.

Request Number: 02h

Source of Data: device

Data Length: 1

Contents of Value field: The high byte is 0. The low byte indicates the report ID that the request applies to. If the low byte is 0, the request applies to all of the device's Input reports.

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: the Idle rate, expressed in units of 4 milliseconds.

Comments: See Set_Idle for more details. HIDs aren't required to support this request.

Set_Idle

Purpose: Saves bandwidth by limiting the reporting frequency of an interrupt IN endpoint when the data hasn't changed since the last report.

Request Number: 0Ah

Source of Data: none

Data Length: 0

Contents of Value field: The high byte sets the duration, or the maximum amount of time between reports. A value of 0 means that there is no maximum and the device will report only when the report data has changed. Otherwise, the device returns a NAK. The low byte indicates the report ID that the request applies to. If the low byte is 0, the request applies to all of the device's Input reports.

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: none

Comments: The duration is in units of 4 milliseconds, which gives a range of 4 to 1,020 milliseconds. No matter what the duration value is, if the report data has changed since the last report sent, on receiving a request, the device sends a report. If the data hasn't changed and the amount of time specified in the duration value hasn't elapsed since the last report, the device returns a NAK. If the data hasn't changed and the amount of time specified in the duration value has elapsed since the last report, the device sends a report. A duration value of 0 indicates an infinite duration; the device sends a report only if the report data has changed, and responds to all other interrupt IN requests with NAK.

HIDs aren't required to support this request. On enumerating a HID, the Windows HID driver attempts to set the idle rate to 0. If the HID supports the request, it will send a report only if the report data has changed. If the HID returns a Stall in response to this request, the request isn't supported and the device can send reports whether or not the data has changed.

Get_Protocol

Purpose: The host learns whether the boot or report protocol is currently active on the device.

Request Number: 03h

Source of Data: device

Data Length: 1

Contents of Value field: 0

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: The protocol. 0=boot protocol, 1=report protocol.

Comments: Boot devices must support this request.

Set_Protocol

Purpose: The host specifies whether to use the boot or report protocol.

Request Number: 0Bh

Source of Data: host

Data Length: 1

Contents of Value field: 0

Contents of Index field: the number of the interface that supports this request.

Contents of data packet in the Data stage: 0=Boot Protocol; 1=Report Protocol

Comments: Boot devices must support this request.

Transferring Data

When enumeration is complete, the host has done all of the following: it has identified the device interface as a HID, it has established pipes with the supported endpoints, and it has learned what report formats to use in sending and receiving data.

The host uses control transfers to send and receive Feature reports containing additional configuration data or other data that doesn't have critical timing requirements. For example, a control-panel application for a video monitor may use control transfers to send settings to the monitor. The host uses interrupt transfers to send and receive periodic, low-latency data in Input and Output reports. The device's firmware must have the complementary code to respond to the host's requests.

Sending Data to the Host

The host receives data after requesting it in an interrupt or control transfer. To respond to an interrupt transfer, the device's firmware needs only to have the requested data in its transmit buffer and to be configured to send the data in response to an interrupt IN request. For Cypress' enCoRe series, doing this requires writing a value to Endpoint 1's transmit configuration register to enable transmitting and to specify the number of bytes to send and the data-toggle bit's value.

Below is example code for the enCoRe that prepares two bytes to transmit on the next interrupt IN transfer:

On receiving a Set_Configuration request, enable the Endpoint 1 interrupt:

```
; Set the endpoint mode to NAK Ins and Outs
mov  A, NAK_IN_OUT
iowr ep1_mode
; Enable Endpoint 0 and 1 interrupts.
mov  A, EP0_INT | EP1_INT
iowr endpoint_int
mov  A, 00h
; Reset the data toggle.
mov  [ep1_data_toggle], A
```

To prepare to send data to the host, copy the data to Endpoint 1's buffer and configure the endpoint to return data in an IN transaction:

```
mov  A, [data_byte_0]
mov  [ep1_dmabuff0], A
mov  A, [data_byte_1]
mov  [ep1_dmabuff1], A
; Configure Endpoint 1 to send 2 bytes.
mov  A, 02h
; Keep the data toggle the same.
or   A, [ep1_data_toggle]
iowr ep1_count
; Configure the endpoint to send data in IN
; transactions.
mov  A, ACK_IN
iowr ep1_mode
```

After sending the data, in Endpoint 1's interrupt service routine, toggle the data toggle so it will be correct for the next transaction:

```
; Toggle the data toggle.
mov  A, 80h
xor  [ep1_data_toggle], A
```

The details will vary for other chips. When the device has no data to send, the endpoint should be configured to return NAK.

Responding to a Get_Report request for a Feature report is much like responding to any control Read request. Control transfers are more complicated than interrupt transfers because of their multiple stages, but you can use the code for other control Read requests as a model. The device must be able to detect the request in the Setup stage, write the requested report data to the USB output buffer for transmitting in the Data stage, and acknowledge the host's 0-length data packet in the Status stage.

Receiving Data from the Host

The host receives data after requesting it in an interrupt or control transfer. As explained earlier, a host may use control or interrupt transfers for Output reports. The chip's architecture and descriptors determine whether or not the HID interface has an interrupt OUT pipe available. The host always uses Set_Report control requests to send Feature reports.

If the interface has an interrupt OUT endpoint and needs to receive low-latency data, the endpoint should be configured to receive report data. Typically, when new data arrives, an interrupt informs the device of the event. An interrupt-service routine in the firmware then does whatever is necessary with the data, either using the data right away or storing it for later use. The interrupt-service routine should also do whatever is needed to pre-pare the endpoint to receive a new report.

If the interface doesn't have an interrupt OUT endpoint, the firmware must detect Set_Report control requests and handle the report data in the requests. The chip must do the same to receive Feature reports. A device that has an interrupt OUT endpoint should also be able to receive reports in Set_Report control transfers so it can receive Feature reports, or Output reports if it happens to communicate with a 1.0 host.

A Set_Report request consists of at least three transactions. The host ini-tiates a Setup transaction that specifies the request and the number of bytes in the report, followed by one or more data transactions with the report data. The device returns a response in the Status stage.

For a Set_Report request, the device must be able to detect the request in the Setup stage, receive the report data in the Data stage, and send a hand-shake in the Status stage. These are the steps a device typically follows to handle a Set_Report request:

1. The device detects a Setup packet, stores the data in the transaction's data packet, returns ACK, and triggers an interrupt that causes the firmware to jump to an interrupt-service routine.

2. The interrupt-service routine does the following:

• Detects the code that indicates the arrival of a Set_Report request.

• Reads the report-length, report-type, and report-ID parameters in the Setup transaction.

• Ensure that Endpoint 0 is configured to accept the data following an OUT token packet.

3. When the interrupt-service routine ends, the device returns to normal operation until it receives an OUT token packet indicating that the host is

sending data to the control endpoint in the Data stage. After receiving the data, the endpoint returns a status code in the handshake packet. An interrupt causes the firmware to jump to an interrupt-service routine for the endpoint.

4. The interrupt-service routine does whatever is needed with the received data.

5. If additional data packets are expected in the Data stage, repeat steps 3 and 4 for any additional OUT transactions, up to the Length value in the Setup transaction.

6. In response to an IN token packet in the Status stage, the endpoint sends a 0-length data packet and the host returns ACK.

Below is enCoRe code that executes on detecting a Set_Report request. The code finds out how many bytes to read and configures Endpoint 0 to receive data in an OUT transaction. This involves setting two configuration bits.

```
set_report:
; Find out how many bytes to read in the OUT
; transaction(s) that will follow.
; This value is in WLengthlo.
; (WLengthhi is unused for this device).
; Save the length in data_count.
mov  A, [wLengthlo]
mov  [data_count], A
mov  A, 0
mov  [wLengthhi], A

; Unlock the counter register so it can be updated
; with the number of bytes in the data stage.
  iord ep0_count

; Enable receiving data in an OUT transaction.
jmp  initialize_control_write

initialize_control_write:
; The firmware uses the value in ep0_transtype to
; decide how to respond to a token packet.
mov  A, TRANS_CONTROL_WRITE
mov  [ep0_transtype], A
```

```
; Set the data toggle.
mov  A, DATA_TOGGLE
mov  [ep0_data_toggle], A

; Send ACK in response to OUT packets,
; which will contain the Control Write data.
; Send NAK in response to IN packets (not expected).
mov  A, ACK_OUT_NAK_IN
iowr ep0_mode

;Return from the Endpoint 0 ISR.
pop  A
pop  X
reti
```

The chip then waits for the arrival of the OUT token packet to begin the Data stage. When an Endpoint 0 interrupt occurs, the code checks for an OUT packet, and if one has arrived, it stores the received data and returns a 0-byte data packet in the Status stage:

```
control_write_data_stage:
; Jump here on receiving an Out packet in the
; Data stage of a Control Write transfer.

; If the data-valid bit isn't set,
; we're done with the data stage.
iord ep0_count
and  A, DATA_VALID
jz   control_write_data_stage_done

; Check the data-toggle bit. If it's incorrect,
; we're done with the Data stage.
iord  ep0_count
and  A, DATA_TOGGLE
xor  A, [ep0_data_toggle]
jnz  control_write_data_stage_done

; Copy the report's bytes to data memory.
mov  A, [ep0_dmabuff0]
mov  [data_byte_0], A
mov  A, [ep0_dmabuff1]
mov  [data_byte_1], A
```

```
;Toggle the data-toggle bit.
mov  A, DATA_TOGGLE
xor  [ep0_data_toggle], A

; Configure Endpoint 0 to send a 0-byte data packet
; in response to an IN packet (the transfer's Status
; stage) and to Stall an Out packet.

mov  A, STATUS_IN_ONLY
iowr  ep0_mode

control_write_data_stage_done:

; Return from Endpoint 0's ISR.
  pop  A
  pop  X
  reti
```

After sending the 0-byte data packet, the endpoint is ready for another transfer.

14

Human Interface Devices: Reports

Chapter 13 introduced the reports that HIDs use to exchange data. A report can be a buffer of undefined bytes, or it can be a complex assortment of items, each with assigned functions and units. This chapter shows how to design a report to fit a specific application.

Report Structure

A report descriptor may contain any of dozens of items arranged in various combinations. It can be long and complex, short and simple, or anywhere in between. The advantage of a more complex descriptor is that the device can provide detailed information about the data it sends and expects to receive. The descriptor can specify the values' uses and what units to apply to the raw data, and it can tell applications whether or not a device supports a particular feature, such as force feedback on a joystick.

But just because the specification supports an item that applies to a device doesn't mean that the report has to include it. For custom devices that are intended for use with a single application, the application often knows the report format in advance, so there's no need to request the information from the device. For example, when the vendor of a data-acquisition unit creates an application for use with the unit, the vendor already knows what data format the device will use in its reports. At most, the application might check the product ID and version number from the device descriptor to learn whether it can request a particular setting or action.

Some of the details about report structures can get tedious, and it's not necessary to understand every nuance about them in most cases. So feel free to skim through the details. You can always come back to them later if you need to.

The report descriptor consists of a series of items that describe the values to be transferred. Each item has a defined scope, and some items may apply to multiple values, eliminating the need to repeat.

Using the HID Descriptor Tool

The HID Descriptor Tool (Figure 14-1) is a free utility available from the USB Implementers Forum. It helps in creating report descriptors, and will also check your descriptor's structure, reporting any errors it finds. Instead of having to look up the values that correspond to each item in your report, you can select the item from a list and enter the value you want to assign to it, and the software will add the item to the descriptor. You can also add items manually. The Parse Descriptor function displays the raw and interpreted values in your descriptor and comments on any errors found. When you have a descriptor with no errors, you can convert it to the syntax required by your firmware. The tool has limited support for vendor-specific items, and may flag these as errors.
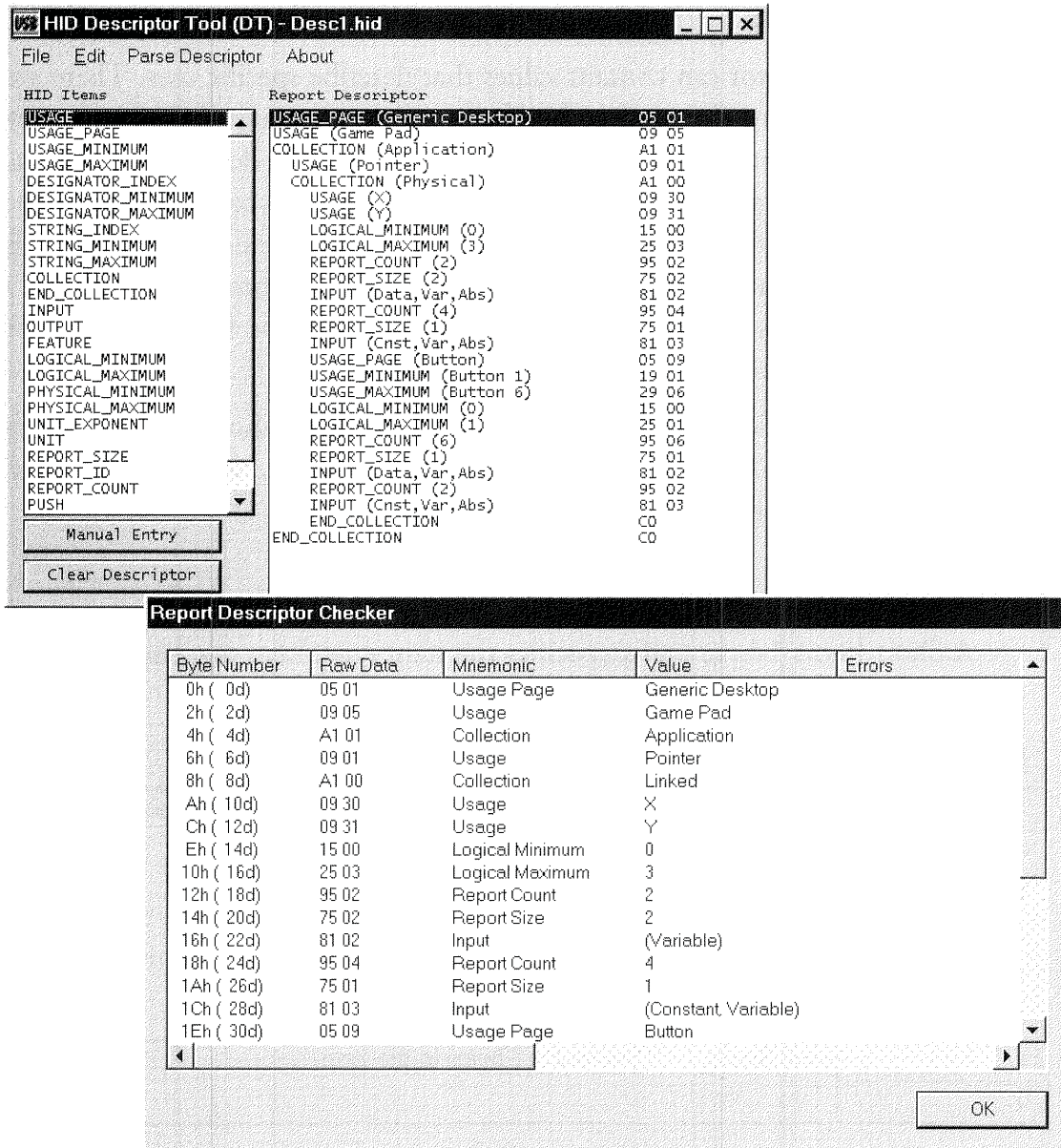
Figure 14-1: The HID Descriptor Tool helps in creating and testing HID report descriptors.

Predefined Values

A report descriptor can contain values that describe specific uses. There are several documents that define the Usage and other values that reports may contain. The first place to look is the *HID Usage Tables* document. This has tables of values for generic desktop controls, simulation controls, game controls, LEDs, buttons, telephony devices, and more. The document also tells you where to find values that are defined elsewhere. Some are in the HID specification, while others are in the class specifications for specific device types such as monitor, power, and image-class devices.

The HID specification defines two report item types: short items and long items. As of HID 1.1, there are no defined Long items, and the type is just reserved for future use.

Short Items

A Short item's 1-byte prefix specifies the item type, item tag, and item size. These are the elements that make up the prefix byte:

Bit Number	Contents	Description
7	Item Tag	Numeric value that indicates the item's function
6		
5		
4		
3	Item Type	Item scope: Main, Global, or Local
2		
1	Item Size	Number of bytes in the item
0		

The item tag (bits 4-7) indicates the item's function.

The item type (bits 3 and 2) describes the scope of the item: Main (00), Global (01), or Local (10). Main items define or group the data fields in the descriptor. Global items describe the reported data. Local items define characteristics of individual controls in the data. (This chapter has more information about these.)

The item size (bits 1 and 0) indicates how many data bytes the item contains. Note that an item size of 3 (11 in binary) corresponds to 4 data bytes:

Item Size (binary)	Number of Data Bytes
00	0
01	1
10	2
11	4

Long Items

A Long item uses multiple bytes to store the same information as the Short item's 1-byte prefix. A Long item's 1-byte prefix (FEh) identifies the item as a Long item. In addition, the item has a byte that specifies the number of data bytes, a byte containing the item tag, and up to 255 bytes of data.

The Main Item Type

A Main item defines or groups data items within a report descriptor. There are five subtypes with the Main item type. The Input, Output, and Feature items each define fields in the report. Collection and End Collection items don't define fields, but instead group related items within a report. The default value for all Main items is 0.

Input, Output, and Feature Items

Table 14-1 shows the supported values for the Input, Output, and Feature items, including the item tag and the meanings of the bits in the value that follows the tag.

An Input item can apply to any control, sensor reading, or other information that the device sends to the host. An Input report contains one or more Input items. The host uses interrupt IN transfers to request Input reports.

An Output item applies to information that the host sends to the device. An Output report contains one or more Output items. Output reports contain data that reports the states of controls, such as whether to open or close a

Table 14-1: The data included with Input, Output, and Feature Item Tags describes the report data.

Main Item Tag	Bit Number	Meaning if bit = 0	Meaning if bit = 1
Input (100000nn, where nn=the number of data bytes)	0	Data	Constant
	1	Array	Variable
	2	Absolute	Relative
	3	No wrap	Wrap
	4	Linear	Non-linear
	5	Preferred state	No preferred state
	6	No null position	Null state
	7	Reserved	
	8	Bit field	Buffered bytes
	9-31	Reserved	
Output (100100nn, where nn=the number of data bytes)	0	Data	Constant
	1	Array	Variable
	2	Absolute	Relative
	3	No wrap	Wrap
	4	Linear	Non-linear
	5	Preferred state	No preferred state
	6	No null position	Null state
	7	Non-volatile	Volatile
	8	Bit field	Buffered bytes
	9-31	Reserved	
Feature (101100nn, where nn=the number of data bytes)	0	Data	Constant
	1	Array	Variable
	2	Absolute	Relative
	3	No wrap	Wrap
	4	Linear	Non-linear
	5	Preferred state	No preferred state
	6	No null position	Null state
	7	Non-volatile	Volatile
	8	Bit field	Buffered bytes
	9-31	Reserved	

switch or the intensity to apply to an effect. As explained earlier, if an interrupt OUT pipe is available, a HID 1.1-compliant host uses interrupt OUT transfers to send Output reports. Otherwise, the host uses Set_Report control requests.

A Feature item normally applies to information that the host sends to the device. However, it's also possible for the host to read Feature items from a device. A Feature report contains one or more Feature items. Feature reports typically contain configuration settings that affect the overall behavior of the device or one of its components. Feature reports normally control settings that you might otherwise adjust in a physical control panel. For example, the host may have a virtual (on-screen) control panel to enable users to select and control features. The host uses control transfers with Set_Report and Get_Report requests to send and receive Feature reports.

Following each item tag are 32 bits that describe the data. At most, only 9 of the bits are used, with the rest reserved. The device firmware and host software may use or ignore this information.

The bit functions are the same for Input, Output, and Feature items, except that Input items don't support the volatile/non-volatile bit. These are the uses for each bit:

Data | Constant. Data means that the contents of the item are modifiable (read/write). Constant means the contents are not modifiable (read-only).

Array | Variable. This bit specifies whether the data reports the state of every control or just reports the controls that are active. Reporting only the active controls results in a more compact report for devices such as keyboards, where there are many controls (keys) but only one or a few are active at the same time.

For example, if a keypad has eight keys, setting this bit to Variable would mean that the keypad's report would contain a bit for each key. In the report descriptor, the report size would be one bit, the report count would be eight, and the total amount of data sent would be eight bits. Setting the bit to Array would mean that each key has an assigned index, and the keypad's report would contain only the index of the keys that are active. With eight keys, the report size would be three bits, which can report a key number

from 0 through 7. The report count would equal the maximum number of simultaneous keypresses that could be reported. If the user can press only one key at a time, the report count would be 1 and the total amount of data sent would be just 3 bits. If the user can press all of the keys at once, the report count would be 8 and the total amount of data sent would be 24 bits.

The specification recommends returning 0 when no controls are active, and specifying a Logical Minimum of 1 and a Logical Maximum equal to the number of controls.

Absolute | Relative. Absolute means that the value is based on a fixed origin; Relative means that the data indicates the change from the last reading. A joystick normally reports absolute data (the joystick's current position), while a mouse reports relative data (how far the mouse has moved since the last report).

No Wrap | Wrap. Wrap indicates that the value rolls over if it continues to increment after reaching its maximum or continues to decrement after reaching its minimum. A value specified as No Wrap that exceeds the limits may report a value outside the specified limits. This bit doesn't apply to Array data.

Linear | Non-linear. Linear indicates that the measured data and the reported value have a linear relationship. A graph of the reported data and the property being measured forms a straight line. In non-linear data, a graph of the reported data and the property being measured forms a curve. This bit doesn't apply to Array data.

Preferred State | No Preferred State. Preferred state indicates that the control will return to a particular state when the user isn't interacting with it. A momentary pushbutton has a preferred state (out) when no one is pressing it. A toggle switch has no preferred state; it remains in the state selected by the last user. This bit doesn't apply to Array data.

No Null Position | Null State. Null state indicates that the control supports a state where it isn't sending meaningful data. A control indicates that it's in its null state by sending a value outside the range defined by its Logical Minimum and Maximum. No Null Position indicates that the control can always be assumed to be sending meaningful data. A hat switch on a joystick

is in a null position when it isn't being pressed. This bit doesn't apply to Array data.

Non-volatile | Volatile. The Volatile bit applies only to Output and Feature reports. Volatile means that the device can change the value on its own, without host interaction, as well as when the host sends a report requesting the device to change the value. For example, a control panel may have a control that users can set in two ways. They may use a mouse to click a setting in a window on the host to cause the host to send a report to the device, or they may press a physical button on the device. Non-volatile means that the device changes the value only when the host requests it in a report.

When the host is sending a report and doesn't want to change a volatile item, the value to assign depends on whether the data is defined as relative or absolute. If a volatile item is defined as relative, a report that assigns a value of 0 should result in no change. If a volatile item is defined as absolute, a report that assigns an out-of-range value should result in no change.

This bit doesn't apply to Array data.

Bit Field | Buffered Bytes. Bit Field means that each bit or a group of bits in a byte can represent a separate piece of data and the field doesn't represent a single quantity. The application interprets the contents of the field. Buffered Bytes means that the data consists of one or more bytes. The report size for Buffered Bytes must be eight. This bit doesn't apply to Array data.

Collection and End Collection Tags

All of the report types can use Collection and End Collection items to group related items.

There are three defined types of collections: application, physical, and logical. Vendors can also define their own collection types. Collections can be nested. Table 14-2 shows the values of the Collection and End Collection tags and the defined values for the different collection types.

An application collection contains items that have a common purpose or together carry out a single function. For example, the boot descriptor for a

Table 14-2: Data values for the Collection and End Collection Main Item Tags.

Main Item Type	Value	Description
Collection (A1h)	00h	Physical
	01h	Application
	02h	Logical
	03h-7Fh	Reserved
	80h-FFh	Vendor-defined
End Collection (C0h)	None	Closes a collection

keyboard groups the keypress and LED data in an application collection. All reports must be in an application collection

A physical collection contains items that represent data at a single geometric point. A device that collects a variety of sensor readings from multiple locations might group the data for each location in a collection. The boot descriptor for a mouse groups the button and position indicators in a physical collection.

A logical collection forms a data structure consisting of items of different types that are linked by the collection. An example is the contents of a data buffer and a count of the number of bytes in the buffer.

Each collection begins with a Collection item and ends with an End Collection item. All Main items between the Collection and End Collection items are part of the collection. Each collection must have a Usage tag (described below).

If a report contains an unknown vendor-defined collection type, the host should ignore all Main items in the collection. If a known collection type has an unknown Usage, the host should ignore all items in the collection.

The Global Item Type

Global items identify reports and describe the data in them, including characteristics such as the data's function, maximum and minimum allowed values, and the size and number of report items. A Global item tag applies to every item that follows until the next Global tag. This saves storage space

because there's no need to repeat values that don't change from one item to the next. There are 12 defined Global items, shown in Table 14-3.

Identifying the Report

Report ID is a prefix that may precede the report data in a data packet. A device can support multiple reports of the same type, with each containing different data and having its own ID. This way, a transfer doesn't have to include every piece of data every time. However, in many cases the simplicity of having a single report is more important than the need to reduce the bandwidth used by reports to the absolute minimum.

In a descriptor, a Report ID item applies to all items that follow until a new Report ID. If there is no Report ID item, the default ID of zero is assumed. A descriptor should not declare a Report ID of zero. Input, Output, and Feature reports can share a Report ID.

If one or more report types has multiple Report IDs, every report must have a declared ID. For example, if an interface supports Report IDs 1 and 2 for Feature reports, any Input or Output reports must also have a Report ID greater than 0.

In a transfer that uses a Set_Report or Get_Report request, the host specifies a report ID in the Setup transaction, in the low byte of the Value field. In an interrupt transfer, if the interface supports more than one report ID, the report ID should be the first byte sent with a report. If the interface supports only the default report ID of zero, the report ID should not be sent with the report in an interrupt transfer.

Under Windows, applications should always precede a report to be sent with a report ID. If the ID is 0, the HID driver doesn't send it on the bus with the report data. In a similar way, reports read into an application always begin with a report ID. The HID driver inserts an ID of zero before the report data if necessary.

When a HID supports multiple report IDs for Input reports of different sizes, Windows' HID driver always uses buffers large enough to hold the longest report. Shorter reports that are not a multiple of the maximum

Table 14-3: There are twelve defined Global items.

Global Item Type	Value (nn indicates the number of bytes that follow)	Description
Usage Page	000001nn	Defines the data's usage or function.
Logical Minimum	000101nn	Smallest value that an item will report.
Logical Maximum	001001nn	Largest value that an item will report.
Physical Minimum	001101nn	The logical minimum expressed in physical units.
Physical Maximum	010001nn	The logical maximum expressed in physical units.
Unit exponent	010101nn	Base 10 exponent of units.
Unit	011001nn	Unit values
Report Size	011101nn	Size of an item's fields in bits.
Report ID	100001nn	Prefix that identifies a report.
Report Count	100101nn	The number of data fields for an item
Push	101001nn	Places a copy of the global item state table on the stack.
Pop	101101nn	Replaces the item state table with the last structure pushed onto the stack.
Reserved	110001nn to 111101nn	For future use.

packet size must terminate with a 0-length data packet to let the host know that all of the data has been sent.

Windows' HID driver uses interrupt transfers to retrieve Input reports. When there are multiple Input Report IDs, the driver has no way to request a specific report. On receiving the IN token packet, the device returns whatever report is in its buffer, so the device firmware must decide which report to make available. The HID driver stores the received report and its ID in its buffer.

Describing the Data's Use

The items that describe how the data will be used are Usage Page, Logical and Physical Maximums and Minimums, Unit, and Unit Exponent. All of these help the receiver of the report to interpret the report's data. All but the Usage Page are involved with converting raw report data to values with units

attached. These items make it possible for a report to contain data in a compact form, with the receiver of the data having the responsibility of converting the data to meaningful values. However, the sender of the report data may instead choose to do some or all of the converting.

Usage Page. An item's Usage is a 32-bit value that describes its function. The Usage is made up of two 16-bit parts: the Usage Page, which is a Global item, and the Usage Index, which is a Local item. Multiple items may share a Usage Page while having different Usage Indexes. After a Usage Page appears in a report, all Usage Indexes that follow will use that Usage Page until a new one is declared. Re-using the Usage Page reduces the amount of data that the descriptor has to store and send.

The HID Usage Tables document lists the defined Usage Pages and their values and also names the document section or other document that describes each page and its indexes. There are Usage Pages for many common device types, including generic desktop controls (mouse, keyboard, joystick), digitizer, bar-code scanner, camera control, and various game controls. Specialized devices may not have a defined Usage Page. In this case, a vendor can define the Usage Page. Values from FF00h to FFFFh are reserved for vendor-defined Usage Pages.

Logical Minimum and Logical Maximum. The Logical Minimum and Maximum define the limits for reported values. The limits are expressed in "logical units," which means that they use the same units as the values they describe. For example, if a device reports readings of up to 500 milliamperes in units of 2 milliamperes, the Logical Maximum is 250.

Negative values may be expressed as two's complements. Bit 7 is a sign bit that indicates whether the value is positive (0) or negative (1). The values 0 to 7Fh are the positive decimal values 0 through 127, and FFh to 80h are the negative decimal values -1 through -128. To find the negative value rep-

resented by a two's complement, complement each bit and add 1 to the result. Here are some examples:

Negative Value Expressed as a Two's Complement:	FFh	FDh	80h
Complement each bit:	00h	02h	7Fh
Add 1:	01h	03h	80h
Value Expressed as a Negative Number (decimal):	-1	-3	-128

The HID specification says that if both the Logical Minimum and Maximum are considered positive, there's no need for a sign bit. For example, a range from 0 to 255 can have a Logical Minimum of 00h and a Logical Maximum of FFh. A device will enumerate and transfer data without problems whether the Logical Minimum and Maximum are expressed as signed or unsigned values. The receiver of the data has to know whether or not the data can be negative.

The HIDView utility (described in Chapter 17) assumes the use of signed values. With a Logical Minimum of 00h and a Logical Maximum of FFh, it reports the error, "Logical Minimum must be less than the Logical Maximum." It doesn't report this error with a minimum of 80h (-128) and maximum of 7F (+127). On the other hand, the HID Descriptor Tool reports an error if you use a minimum of 80h and maximum of 7Fh, while it accepts 00h and FFh.

The Physical Minimum, Physical Maximum, Unit Exponent, and Unit items define how to convert the reported values into more meaningful units.

Physical Minimum and Physical Maximum. The Physical Minimum and Maximum define the limits for the value when expressed in the units defined by the Units tag. In the earlier example of values of 0 through 250 in units of 2 milliamperes, the Physical Minimum is 0 and the Physical Maximum is 500. The receiving device uses the logical and physical limit values to obtain the value in the desired units. In the example, reporting the data in units of 2 milliamperes means that the value can transfer in a single byte, with the receiver of the data using the Physical Minimum and Maximum values to translate to milliamperes. The price is a loss in resolution,

compared to reporting 1 bit per milliampere. If the report doesn't specify the values, they default to the same as the Logical Minimum and Maximum.

Unit Exponent. The Unit Exponent specifies what power of 10 to apply to the value obtained after using the logical and physical limits to translate the value into the desired units. The exponent can range from -8 to +7. A value of 0 causes the value to be multiplied by 10^0, or 1, which is the same as applying no exponent. These are the codes:

Exponent	0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1
Code	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh

For example, if the value obtained is 1234 and the Unit Exponent is 0Eh, the final value is 12.34.

Unit. The Unit tag specifies what units to apply to the report data after it's converted using the Physical and Unit Exponent items. The HID specification defines codes for the basic units of length, mass, time, temperature, current, and luminous intensity. Most other units can be derived from these.

Specifying a Unit value can be more complicated than you might expect. Table 14-4 shows values you can work from. The value can be as long as four bytes, with each nibble having a defined function. Nibble 0 (the least significant nibble) specifies the measurement system, either English or SI (International System of Units), and whether the measurement is in linear or angular units. Each of the nibble positions that follow represents a quality to be measured, with the value of the nibble representing the exponent to apply to the value. For example, a nibble with a value of 2 means that its corresponding value is in units squared. A nibble with a value of 0Dh, which represents -3, means that the units are expressed as 1/units^3. These exponents are separate from the Unit Exponent value, which is a power of ten applied to the data, rather than an exponent applied to the units.

Converting Raw Data

To convert raw data to values with units attached, three things must occur. The firmware's report descriptor must contain the information needed for

Table 14-4: The units to apply to a reported value are a function of the measuring system and exponent values specified in the Unit item

Nibble Number	Quality Measured	Measuring System (Nibble 0 value)				
		None (0)	SI Linear (1)	SI Rotation (2)	English Linear (3)	English Rotation (4)
1	Length	None	Centimeters	Radians	Inches	Degrees
2	Mass	None	Grams		Slugs	
3	Time	None	Seconds			
4	Temperature	None	Fahrenheit		Celsius	
5	Current	None	Amperes			
6	Luminous Intensity	None	Candelas			
7	Reserved	None				

the conversion. The sender of the data must send data that matches the specification in the descriptor. And the receiver of the data must apply the conversions specified in the descriptor.

Below are examples of descriptors and raw and converted data. Remember that just because a tag exists in the HID specification doesn't mean you have to use it. If the application knows what format and units to use for the values it's going to send or receive, the firmware doesn't have to specify it.

To measure time in seconds, up to a minute, the report descriptor might include this information:

> Logical Minimum: 0
>
> Logical Maximum: 60
>
> Physical Minimum: 0
>
> Physical Maximum: 60
>
> Unit: 1003h. Nibble 0 = 3 to select the English Linear measuring system (though in this case, any value from 1 to 4 would work). Nibble 3 = 1 to select time in seconds.
>
> Unit Exponent: 0

With this information, the receiver knows that the value sent equals a number of seconds.

Now, what if instead you want to measure time in tenths of seconds, again up to a minute? You would need to increase the Logical and Physical Maximums and change the Unit Exponent:

> Logical Minimum: 0
>
> Logical Maximum: 600
>
> Physical Minimum: 0
>
> Physical Maximum: 600
>
> Unit: 1003h. Nibble 0 = 3 to select the English Linear measuring system. Nibble 3 = 1 to select time in seconds.
>
> Unit Exponent: 0Fh. This represents an exponent of -1, to indicate that the value is expressed in tenths of seconds rather than seconds.

Sending values as large as 600 will require 3 bytes, which the firmware specifies in the Report Size tag.

To send a temperature value using one byte to represent temperatures from -20 to 110 degrees Fahrenheit, the report descriptor might contain the following:

> Logical Minimum: -128 (80h expressed as a two's complement)
>
> Logical Maximum: 127 (7Fh)
>
> Physical Minimum: -20 (ECh expressed as a two's complement)
>
> Physical Maximum: 110 (6Eh)
>
> Unit: 10003h. Nibble 0 is 3 to select the English Linear measuring system, though in this case, any value from 1 to 4 is OK. Nibble 4 is 1 to select degrees Fahrenheit.
>
> Unit Exponent: 0

These values ensure the highest possible resolution, because the transmitted values can span the full range from 0 to 255.

In this case the logical and physical limits differ, so converting is required. To find the resolution, or number of bits per unit, use this equation:

```
Resolution = _
  (Logical_Maximum - Logical_Minimum) / _
  ((Physical_Maximum - Physical_Minimum) * _
  (10 ^ Unit_Exponent ))
```

With the example values, this works out to 1.96 bits per degree, or 0.51 degree per bit.

To convert a value to the specified units, use this equation:

```
Value = _
   Value_In_Logical_Units *
   ((Physical_Maximum - Physical_Minimum) * _
   (10 ^ Unit_Exponent )) /
   (Logical_Maximum - Logical_Minimum)
```

If the value in logical units (the raw data) is 63, the converted value in the specified units is 32 degrees Fahrenheit.

Specifying velocity in centimeters per second requires a Unit value that contains units of both centimeters and seconds. From Table 14-4, the Unit value to use is 1011h. Nibble 0 = 1 to select the SI measuring system, nibble 1 = 1 to select length in centimeters, and nibble 3 = 1 to select time in seconds.

To illustrate how complicated it can get, the Unit value for volts is F0D121h, which indicates the SI Linear measuring system in units of $(cm^2)*(gm)/(sec^{-3})*(amp^{-1})$. However, remember that the Unit value only specifies the units. All the receiver has to do is identify the Units value and assign the units to received data; there's no need to do the calculations implied in the Units value.

Describing the Data's Size and Format

Two Global items describe the size and format of the report data.

Report Size specifies the size in bits of an Input, Output, or Feature item's fields. Each field contains one piece of data.

Report Count specifies how many fields an Input, Output, or Feature item contains. For example, for two 8-bit fields, Report Size is 8 and Report Count is 2. For ten 4-bit fields, Report Size is 4 and Report Count is 10. For one 16-bit field, Report Size is 16 and Report Count is 1.

A single Input, Output, or Feature report can have multiple items, each with its own Report Size and Report Count.

Saving and Restoring Global Items

The final two Global items enable saving and restoring sets of Global items. These allow flexibility in the report formats while using minimum storage space in the device.

Push places a copy of the Global-item state table on the CPU's stack. The Global-item state table contains the current settings for all previously defined Global items.

Pop is the complement to Push. It restores the saved states of the previously pushed Global item states.

The Local Item Type

Local items define qualities of the knobs, switches, buttons, and other controls that a report returns data for. A Local item applies to all controls that follow within the Main item, until a new value is assigned. Local items don't carry over to the next Main item. Each Main item begins fresh, with no Local items defined.

Local items relate to general usages, body-part designators, and strings. A Delimiter item enables grouping sets of Local items. Table 14-5 shows the values and meaning of each of the items.

Usage. The Local Usage item is the Usage Index that works together with the Global Usage Page to describe the function of an item or collection. As with the Usage Page, the HID Usage Tables document lists many Usage Indexes. For example, the Buttons Usage Page uses Local Usage Indexes from 1 to FFFFh to specify individual buttons, with a value of 0 meaning no button pressed.

A report may assign one Usage to multiple controls, or it may assign a different Usage to each control. If a report item is preceded by a single Usage, that Usage applies to all of the item's controls. If a report item is preceded by more than one Usage, and the number of controls equals the number of Usages, each Usage applies to one control, with the Usages and controls

Table 14-5: There are ten defined Local items.

Local Item Type	Value (nn indicates the number of bytes that follow)	Description
Usage	000010nn	An index that describes the use for an item or collection.
Usage Minimum	000110nn	The starting Usage associated with an array or bitmap.
Usage Maximum	001010nn	The ending Usage associated with an array or bitmap.
Designator Index	001110nn	Designates the body part used for a control.
Designator Minimum	010010nn	The starting Designator associated with an array or bitmap.
Designator Maximum	010110nn	The ending Designator associated with an array or bitmap.
String Index	011110nn	Associates a string with an item or control.
String Minimum	100010nn	The first string index when assigning a group of sequential strings to controls in an array or bitmap.
String Maximum	100110nn	The last string index when assigning a group of sequential strings to controls in an array or bitmap.
Delimiter	101010nn	The beginning (1) or end (0) of a set of Local items.
Reserved	101011nn to 111110nn	For future use.

pairing up in sequence. In the following example, the report contains two bytes. The first byte's Usage is X, and the second byte's Usage is Y.

```
Report Size (8),
Report Count (2),
Usage (X),
Usage (Y),
Input (Data, Variable, Absolute),
```

If a report item is preceded by more than one Usage and the number of controls is greater than the number of Usages, each Usage pairs up with one control, and the final Usage applies to all of the remaining controls. In the following example, the report is 16 bytes. Usage X applies to the first byte,

Usage Y applies to the second byte, and a vendor-defined Usage applies to the third through 16th bytes.

```
Usage (X)
Usage (Y)
Usage (vendor defined)
Report Count (16),
Report Size (8),
Input (Data, Variable, Absolute)
```

Usage Minimum and Maximum. The Usage Minimum and Maximum can assign a single Usage to multiple controls. The following example reports the state (0 or 1) of each of three buttons. The Usage Minimum and Maximum assign the Button Usage Page to all three items. The item uses one bit per button.

```
Logical Minimum (0)
Logical Maximum (1)
Report Count (3)
Report Size (1)
Usage Page (Button Page)
Usage Minimum (1)
Usage Maximum (3)
Input (Data, Variable, Absolute)
```

The Usage Minimum and Maximum can also assign a single Usage to a series of array items.

Designator Index. For items with a Physical descriptor, the Designator Index specifies the body part the control uses.

Designator Minimum and Maximum. When a report contains multiple controls with the same Designator, the Designator Minimum and Maximum can specify which controls the Usage applies to.

String Index. An item or control can include a string index to associate a string with that item or control. The strings are stored in the same format described in Chapter 5 for product, manufacturer, and serial-number strings.

String Minimum and Maximum. When a report contains multiple controls with the same String Index, the String Minimum and Maximum can specify which controls the Usage applies to.

Delimiter. The Delimiter defines the beginning (1) or end (0) of a local item. A delimited local item may contain alternate usages for a control. This enables different applications to define a device's controls in different ways. For example, a button may have a generic use (Button1) and a specific use (Send, Quit, etc.).

Physical Descriptors

A physical descriptor describes the part or parts of the body intended to activate a control. For example, each finger might have its own assigned control.

A physical descriptor is a type of class descriptor. The host can retrieve a physical descriptor by sending a Get_Descriptor request with 23h in the high byte of the Value field and 00h in the low byte of the Value field.

Physical descriptors are optional. For most devices, they either don't apply at all or the information they could provide has no practical use. The HID specification has more information on how to use physical descriptors, for those devices that need them.

Padding

To pad a descriptor so it contains a multiple of eight bits, the descriptor may include a Main item with no assigned Usage. The following example describes an Input report that transfers three bits with data and five bits of padding:

```
Report Count (3)
Report Size (1)
Usage Page (Button Page)
Usage Minimum (1)
Usage Maximum (3)
Input (Data, Variable, Absolute)
Report Size (5),
Input (Constant)
```

15

Human Interface Devices: Host Application Primer

Chapter 13 and Chapter 14 described human-interface-device communications from the device's perspective and the report format that HIDs use to exchange data with the host. This chapter introduces the Windows functions that applications can use to communicate with HIDs. Applications may use any programming language that can call API functions. Chapter 16 has example code in Visual Basic and Visual C++. Much of the information in this chapter applies to communicating with any USB device, not just HIDs.

Host Communications Overview

Windows 98 and Windows 2000 include everything applications need to communicate with HID-class devices. There's no need to install drivers because Windows has them built in.

How the Host Finds a Device

Communicating with a HID isn't as simple as opening a port, setting a few parameters, and then reading and writing data, as you can do with RS-232 and parallel ports. Before an application can exchange data with a HID, it has to identify the device and get information about its reports. To do this, the application has to jump through a few hoops by calling a series of API functions. The application first finds out what HIDs are attached to the system. It then examines information about each until it finds one with the desired attributes. For a custom device, the application can search for specific Vendor and Product IDs. Or the application can search for a device of a particular type, such as a mouse or joystick.

After finding a device, the application can exchange information with it by sending and receiving reports.

Table 15-1 lists API functions used in establishing communications and exchanging data with a HID. The functions are listed in a typical order that an application might call them.

Table 15-1: Communicating with HIDs uses a variety of API functions. These are the major functions used in identifying a HID and sending and receiving reports.

API Function	DLL	Purpose
HidD_GetHidGuid	hid.dll	Obtain the GUID for the HID class
SetupDiGetClassDevs	setupapi.dll	Return a device information set containing all of the devices in a specified class.
SetupDiEnumDeviceInterfaces	setupapi.dll	Return information about a device in the device information set.
SetupDiGetDeviceInterfaceDetail	setupapi.dll	Return a device pathname.
SetupDiDestroyDeviceInfoList	setupapi.dll	Free resources used by SetupDiGetClassDevs.
CreateFile	kernel32.dll	Open communications with a device.
HidD_GetAttributes	hid.dll	Return a Vendor ID, Product ID, and Version Number.
HidD_GetPreparsedData	hid.dll	Return a handle to a buffer with information about the device's capabilities
HidP_GetCaps	hid.dll	Return a structure describing the device's capabilities.
HidD_FreePreparsedData	hid.dll	Free resources used by HidD_GetPreparsedData.
WriteFile	kernel32.dll	Send an Output report to the device.
ReadFile	kernel32.dll	Read an Input report from the device.
HidD_SetFeature	hid.dll	Send a Feature report to the device.
HidD_GetFeature	hid.dll	Read a Feature report from the device.
CloseHandle	kernel32.dll	Free resources used by CreateFile.

Documentation

The functions are in three DLLs whose documentation is spread among several areas in the Windows DDK documentation and the MSDN library. These are DLLs that contain functions used in HID communications:

Filename	Type of Functions Included
hid.dll	HID communications.
setupapi.dll	Finding and identifying devices
kernel32.dll	Exchanging data, other general functions

The functions that relate only to HID communications are in *hid.dll* and are documented in the DDK, under *Kernel-Mode Drivers > Drivers for Input Devices.* Functions related to detecting devices are in *setupapi.dll* and are documented in the DDK under *Setup, Plug & Play, and Power Management > Device Installation Functions* and also in the Platform SDK under *Device Management Functions.* These functions apply to all Plug-and-Play devices, including USB devices. Functions relating to opening communications, reading Input reports, and writing Output reports are in *kernel32.dll* and are documented in the MSDN library, in the Platform SDK under *File I/O.* Many other devices also use these functions.

Windows 98 SE added seven HID functions to those supported by Windows 98 Gold. Windows 2000 and Windows Me support the new functions as well. The Windows 2000 DDK documentation includes the added functions; the Windows 98 DDK doesn't.

The HID Functions

Hid.dll supports many more functions than the essentials listed in Table 15-1. The following three tables together comprise a complete list of the HID functions grouped by purpose. Functions whose names begin with HidP are available to both applications and device drivers. Functions whose names begin with HidD are available only to applications.

Table 15-2 lists functions that applications use to learn about a HID. Table 15-3 lists functions that applications use in reading and writing reports. Table 15-4 lists functions that applications use in configuring the input buffer to receive reports. The documentation also names three functions for future use: HidD_GetConfiguration, HidD_SetConfiguration, and HidP_TranslateUsagesToI8042ScanCodes.

You can use these functions with just about any HID-class device, including custom designs. Windows 2000 doesn't allow applications to use the functions to access the system keyboard or mouse, but applications don't normally need to do so because the operating system provides other ways to communicate with the keyboard and mouse.

Table 15-2: Applications can use these functions in *hid.dll* to learn about a device.

Function	Purpose
HidD_GetAttributes	Retrieves the HID's Vendor ID, Product ID, and Version Number.
HidD_FreePreparsedData	Frees resources used by HidD_GetPreparsedData.
HidD_GetHidGuid	Obtains the GUID for the HID class.
HidD_GetIndexedString*	Retrieves a string identified by an index.
HidD_GetManufacturerString*	Retrieves the string that identifies the device manufacturer.
HidD_GetPhysicalDescriptor*	Retrieves the string that identifies the physical device.
HidD_GetPreparsedData	Retrieves a handle to a buffer with information about the device's capabilities.
HidD_GetProductString*	Retrieves the string that identifies the product.
HidD_GetSerialNumberString*	Retrieves the string containing the device's serial number.
HidP_GetButtonCaps	Retrieves the capabilities of all buttons in a report.
HidP_GetCaps	Retrieves a pointer to a structure describing the device's capabilities.
HidP_GetLinkCollectionNodes	Retrieves an array of structures that describes the relationship of link collections within a top-level collection.
HidP_GetSpecificButtonCaps	Retrieves the capabilities of buttons in a report. The request can specify a Usage Page, Usage, or Link Collection.
HidP_GetSpecificValueCaps	Retrieves the capabilities of values in a report. The request can specify a Usage Page, Usage, or Link Collection.
HidP_GetValueCaps	Retrieves the capabilities of all values in a report.
HidP_MaxUsageListLength	Retrieves the maximum number of buttons that a report can return. Can specify a Usage Page.
HidP_UsageListDifference	Compares two button lists and find the buttons that are set in one list and not in the other.
*not supported under Windows 98 Gold.	

DirectX

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 with a goal of providing fast access to hardware. Instead of having to poll an

Table 15-3: Applications can use these functions in *hid.dll* to read and write reports.

Function	Purpose
HidD_GetFeature	Retrieves a Feature report.
HidD_SetFeature	Sends a Feature report.
HidP_GetButtons	Returns a pointer to a buffer containing the Usage of each button that is pressed. Can specify a Usage Page.
HidP_GetButtonsEx	Returns a pointer to a buffer containing the Usage and Usage Page of each button that is pressed.
HidP_GetScaledUsageValue	Returns the signed result of a value that has been adjusted for its scaling factor.
HidP_GetUsageValue	Returns a pointer to a value.
HidP_GetUsageValueArray	Returns data for a Usage that contains multiple data items.
HidP_SetButtons	Sets button data.
HidP_SetScaledUsageValue	Takes a signed, physical (scaled) number, converts it to the logical representation used by the device, and inserts it in a report.
HidP_SetUsageValue	Sets a value.
HidP_SetUsageValueArray	Sets data for a Usage that contains multiple data items.

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

The DirectInput components of DirectX enable communications with HIDs under C++, Delphi, or Visual Basic. The DirectX SDK has examples in Visual C++ and Visual Basic. The samples are oriented towards communicating with standard device types. The documentation suggests that you can use DirectX to communicate with any HID, but provides few details on how to do so.

Using API Functions

The examples in this chapter use Microsoft's Visual Basic and Visual C++. As explained in Chapter 10, an API function is a part of Windows' Application Programmer's Interface, which contains thousands of functions that applications can use to communicate with the operating system. The execut-

Table 15-4: Applications can use these functions in *hid.dll* to control the driver's input buffer for reading reports.

Function	Purpose
HidD_FlushQueue*	Empty the input buffer.
HidD_GetNumInputBuffers*	Retrieves the size of the ring buffer the driver uses to store input reports. The default is 8.
HidD_SetNumInputBuffers*	Sets the size of the ring buffer the driver uses to store input reports.
*Not supported under Windows 98 Gold.	

able code for the functions resides in dynamic linked library (DLL) files provided with Windows.

Before getting into the details of the functions themselves, I'll present some background on how to call API functions from Visual Basic and Visual C++ applications. If you're already familiar with using API calls, or if you want to get right to the HID-specific functions, you can skip over the these introductory sections. I'll begin with Visual C++.

Using Visual C++

To use an API function, a Visual C++ application needs three things: the ability to locate the file containing the function's compiled code, a function declaration, and a call that causes the function to execute.

Applications that access HIDs will call functions contained in *hid.dll* and *setupapi.dll*. Each of the DLLs has two companion files, a library file (*hid.lib* and *setupapi.lib*) and one or more header files (*hidpi.h*, *hidsdi.h*, *hidusage.h*, and *setupapi.h*). The header file contains the prototypes, structures, and symbols for the functions that applications may call, and the library file eliminates the need for the application to get a pointer to the function in the DLL.

A DLL contains compiled code for the functions that it exports, or makes available to applications. For each exported function, the DLL's library file contains a stub function whose name and arguments match the name and arguments of one of the DLL's functions. The stub function calls its corresponding function in the DLL. During the compile process, the linker

incorporates the code in the library file into the application's executable file. When the application calls a function in the library file, the function of the same name in the DLL executes.

The *hid.dll* and *setupapi.dll* files are included with Windows. They're typically stored in the *windows\system* or *windows\system32\drivers* folder. (In Windows 2000, substitute *winnt* for *windows*.) Both are standard locations that Windows searches when DLL functions are called. The library and header files are included in the DDK.

The header files for other common Windows functions are included automatically when you create a project. For example, *afxwin.h* adds headers for common Windows and MFC functions.

To include a API function in an application, you need to do the following:

1. Add the library files to the project. In Visual C++, click Project > Settings > Link > Category: Input. In the *Object/library* modules box enter *hid.lib* and *setupapi.lib*. In the same window, if necessary, you can enter a path for the library files under *Additional library path*.

2. Include the header files in one of the application's files. Here's an example:

```
extern "C" {
#include "hidsdi.h"
#include <setupapi.h>
}
```

The #include directive causes the contents of the named file to be included in the file, the same as if they were copied and pasted into it.

The extern "C" modifier enables a C++ module to include header files that use C naming conventions. The difference is that C++ uses name decoration, or name mangling, on external symbols. The punctuation around the file name determines where the compiler will search for the file, and in what order. This is relevant if you have different versions of a file in multiple locations!

Enclosing the file name in brackets (<setupapi.h>) causes the compiler to search for the file first in the path specified by the compiler's */I* option, then in the paths specified by the Include environment variable. Enclosing the

file name in quotes ("hidsdi.h") causes the compiler to search for the file first in the same directory as the file containing the #include directive, then in the directories of any files that contain #include directives for that file, then in the path specified by the compiler's /I option, and finally in the paths specified by the Include environment variable.

3. Call the function. Here is code that declares the variable HidGuid and passes a pointer to it in the function HidD_GetHidGuid in *hid.dll*:

```
GUID     HidGuid;
HidD_GetHidGuid(&HidGuid);
```

Using Visual Basic

In Visual Basic, the process of calling API functions is different than in Visual C++. In place of an include file, the application needs a module containing Visual-Basic declarations for the DLL's functions and structures. Some of these, but not all, are provided with Visual Basic. You don't need library files, as Visual Basic requires only the DLL's name and the DLL itself in a standard or specified location.

You can write a lot of Visual-Basic applications without ever coding an API call. Visual Basic provides its own syntax and controls for performing common functions. For example, to print a file, you can use Visual Basic's Printer Object instead of API functions. The Printer Object provides an easier and more fail-safe way to access printers. When you run the application, the code that executes may call API functions, but Visual-Basic programmers are insulated from having to make the calls directly.

But sometimes you may want to do something that Visual Basic doesn't support explicitly. In these cases, which can include communicating with HIDs, Visual-Basic applications can call API functions.

In a Visual-Basic application, the code to call an API function follows the same syntax rules as the code to call any function. But instead of placing the function's executable code in a routine within the application, the API function requires only a declaration that enables Windows to find the DLL containing the function's code.

Calling API functions in Visual Basic requires some extra knowledge. The documentation included with Visual Basic doesn't offer much guidance. Microsoft's documentation for the API functions uses C syntax to show how to declare and call the functions. The DDK includes the declarations in header files that Visual C++ programmers can include in applications. To use an API function in Visual Basic, you need to translate the declaration and function call from C to Visual Basic.

The process is more complicated than a simple word-for-word translation, mainly because Visual Basic doesn't support all of C's structures, and it stores string variables in a different format. Before you can translate, you need to understand exactly what the function is passing and returning. Even if you have an example to work from, understanding what the function is doing helps in using it correctly.

For greater detail on API calls in Visual Basic, I recommend Dan Appleman's books, especially *Dan Appleman's Win32 API Puzzle Book and Tutorial for Visual Basic Programmers*. This is the book I used as a reference in figuring out how to call the API functions in this chapter.

To use an API function in a Visual Basic program, you need three things: the DLL containing the function, a declaration that enables the application to find and use the function, and a call that causes the function to execute.

The Declaration

This is a Visual-Basic declaration for the API function WriteFile, which you can use to write data to a HID (as well as to files and other devices):

```
Public Declare Function WriteFile _
    Lib "kernel32" _
    (ByVal hFile As Long, _
    ByRef lpBuffer As Byte, _
    ByVal nNumberOfBytesToWrite As Long, _
    ByRef lpNumberOfBytesWritten As Long, _
    ByVal lpOverlapped As Long) _
As Long
```

The declaration includes several pieces of information:

* The function's name (WriteFile).

- The values the function will pass to the operating system (hFile, lpBuffer, nNumberOfBytesToWrite, lpNumberOfBytesWritten, and lpOverlapped). The names use the convention of adding a prefix to indicate the type of data the variable contains: h=handle, lp=long pointer, and so on.

- The data types of the values passed (Long, Byte).

- Whether the values will be passed by value (ByVal) or by reference (ByRef).

- The name of the file that contains the executable code for the function (*kernel32.dll*).

- The data type of the value returned for the function (Long). A few API calls have no return value and may be declared as subroutines rather than functions.

The declaration must be in the Declarations section of a module. You might want to place the declarations for API functions and the user-defined types they pass in a separate module (a *.bas* file) in your project. This will make them easy to add to multiple projects.

Visual Basic's documentation includes the file *win32api.txt*, which contains declarations for many API calls. You can add this file as a module in your project, or you can cut and paste the declarations you need into another module in the project. However, the file doesn't include every API call, especially newer ones like those that relate to HID communications.

To declare a function not included in *win32api.txt*, the starting point is Microsoft's documentation, which includes a declaration in C, comments, and sometimes an example. You can also find C declarations in the header files included in the DDKs. Sometimes these header files have useful comments as well. The header files are text files that you can view in any word processor.

These are header files that have HID-related declarations:

File Name	Contents
hid.h	HID user-mode declarations and functions
hidpi.h	Public interface to the HID parsing library
hidsdi.h	Public definitions for the code that implements the HID DLL
hidusage.h	HID usages
setupapi.h	Setup services

Sometimes the function's documentation names the header file. If not, a quick way to find it is to use the *Find > Files or Folders* utility available from Windows' *Start* menu. In the *Named* text box, enter **.h*, and in the *Containing Text* text box, enter the name of the function whose declaration you want to find. Be sure that *Include Subfolders* is checked, and let Windows go to work finding the file for you.

In some cases, the translation from C to Visual-Basic syntax is fairly straightforward. In others, the C parameters don't correspond in a simple way to the alternatives in Visual Basic.

These are some general guidelines for creating Visual-Basic declarations:

Variable Types

C and Visual Basic each use different terms to specify variable types, and C supports more variable types than Visual Basic. However, to specify a variable type for an API call, all you really have to do is determine the variable's

length, then use a Visual-Basic type that matches. These are some of the C types and their Visual-Basic equivalents:

C Type	Visual-Basic Type
CHAR	Byte
USHORT USAGE	Integer
ULONG HWND BOOLEAN DWORD LP_ (long pointer prefix) P_ (long pointer prefix)	Long
PCTSTR	String

To avoid problems that can result from passing the wrong variable type, an API declaration should declare variables as specific types if possible. In some cases, an application may use a variable in multiple ways, each requiring a different type. There are two ways to handle this. You can create multiple declarations, using the Alias keyword to give each a different name, or you can declare the variable As Any and specify the variable type in the function call.

ByRef and ByVal

For each variable, you have a choice of passing it by reference (ByRef) or by value (ByVal). These parameters have the same meanings as when you use them in the functions and subroutines you write in Visual-Basic applications. Often either will work. But the concept is important to understand when calling API functions, because many of the functions have variables that must be passed a specific way.

ByRef and ByVal determine what information the call passes to enable the function to access the variable. Every variable has an address in memory where its value is stored. When an application passes a variable to a function, it can pass the variable's address or the value itself. The information is passed by placing it on the stack, which is a temporary storage location used (among other things) to pass values to functions.

Passing a variable ByRef means that the function call places the address of the variable on the stack. If the function changes the value by writing a new value to the address, the new value will be available to the calling application because the new value will be stored at the address where the application expects to find it. The address passed is called a pointer, because it points to, or indicates, the address where the value is stored.

Passing a variable ByVal means that the function call places the value of the variable on the stack. The value at the variable's original address in memory is unchanged. If the function changes the value, the calling application won't know about it because the function has no way to pass the new value back to the application.

Passing ByRef is the default, but you can include the ByRef parameter in declarations if you wish. This way, you can quickly see if you've forgotten to assign the parameter to a value. If the declaration doesn't include ByVal or ByRef, you can specify either when you call the function.

For all variable types except strings, there are two situations where you must pass a variable ByRef:

- The called function changes the value and the calling application needs to use the new value. Passing ByRef enables the calling application to access the new value.

- The variable is a user-defined type. You can't pass user-defined types ByVal in Visual Basic.

String variables are a special case. Visual Basic uses a format called BSTR for storing strings in memory. The BSTR format differs from the format expected by API calls. In memory, a BSTR string consists of four bytes containing the string's length in bytes followed by the string's characters in Unicode (2 bytes per character). In contrast, most Windows 98 API functions expect a string to consist of a series of ANSI character codes (1 byte per character), followed by a null (0) termination. Windows 2000 supports two versions of most functions, one that uses Windows 98's ANSI format and one that uses Unicode characters followed by a null termination.

Fortunately, there is a solution that doesn't require the application code to translate between formats. If the string is declared ByVal, Visual Basic creates a copy of the string in ANSI format and passes a pointer to the string. In other words, declaring a Visual-Basic string ByVal actually causes the string to be passed ByRef in the expected format. If the function will change the contents of the string, the application should initialize the string to be at least as long as the longest expected returned string.

For various reasons, some structures can't be passed either ByRef or ByVal. In these cases, there is an alternate way. It requires creating a byte array equal to the structure's size, then using Visual Basic's undocumented VarPtr operator to pass the byte array's address ByVal. When the function returns, the application can copy the data from the byte array into a structure, which is a user-defined variable type.

Passing Nulls

When an optional parameter is a pointer, a function may accept a null value (zero) to indicate that the function call isn't using the pointer.

For example, CreateFile includes a parameter that points to a security-attributes structure. The parameter is declared ByRef:

```
ByRef lpSecurityAttributes As SECURITY_ATTRIBUTES
```

If the call isn't using security attributes, the application should pass zero. But if you pass a value of zero ByRef, the function actually passes the address of a memory location that contains zero. Windows 98 handles the call without error, but Windows 2000 returns *Invalid access to memory location.*

In Visual C++, the solution is to pass a NULL constant. In Visual Basic, declare the parameter ByVal as a Long:

```
ByVal lpSecurityAttributes As Long
```

Then pass a value of 0 in the function call.

If a parameter is declared As Any and you want to pass a Long, use a trailing & (for example, 0&) to ensure that the value is passed as a Long.

Functions and Subroutines

Most API routines are functions, which have a return value that the declaration must also specify. A few are subroutines, with no return value. You can declare these as subroutines, or as functions with the returned value ignored.

Providing the DLL's Name

Each declaration must also name the file that contains the function's executable code. The file is a DLL. When the application runs, Windows loads the named DLLs into memory (unless they're already loaded).

In most cases, the declaration only has to include the file name and not the location. The DLLs used for HID communications are included with Windows. When the first HID enumerates on the system, the DLLs are stored in standard locations (such as \windows\system) that the operating system searches automatically. The operating system also searches the application's working directory for a DLL. In the Visual-Basic environment, the working directory is Visual Basic's directory, not your application's directory. If you use a DLL that isn't stored in a standard Windows directory or the application's working directory, the declaration must specify the location.

For some system files, such as kernel32, the .dll extension is optional in the declaration.

Strings

As mentioned earlier, Windows 98 and Windows 2000 differ in how they store strings. Windows 98 stores each character as an 8-bit ANSI code, while Windows 2000 stores each character as a 16-bit Unicode. To handle the difference, there are two versions of API calls that pass string variables. The 8-bit version ends in A (ANSI), and the 16-bit version ends in W (wide). For example, there is a SetupDiGetClassDevsA function and a SetupDiGetClassDevsW function.

Both Windows 98 and Windows 2000 support the ANSI versions. Windows 98 supports very few Unicode functions. Windows 2000 uses Unicode internally, but can convert to and from ANSI as needed.

Structures

Some of the API functions used in HID applications pass and return structures, which contain multiple items that may be of different types. The documentation for the API functions includes documentation for the structures used by the calls. The header files contain declarations for the structures in C syntax.

Here again, Visual Basic uses different syntax and translating is required. In Visual Basic, you can declare structures as user-defined types. Some of the structures translate in a straightforward way. For example, the Visual-Basic declaration for the HIDD_ATTRIBUTES structure consists of Long and Integer variables that translate directly from the USHORT and ULONG types in the C declaration:

```
Public Type HIDD_ATTRIBUTES
    Size As Long
    VendorID As Integer
    ProductID As Integer
    VersionNumber As Integer
End Type
```

You can then declare a variable of the user-defined type:

```
Dim DeviceAttributes As HIDD_ATTRIBUTES
```

Before passing the structure in an API call, the Size property must be set to the size of the structure in bytes. The LenB operator will do this:

```
DeviceAttributes.Size = LenB(DeviceAttributes)
```

The HidD_GetAttributes API function can then pass the structure ByRef:

```
Public Declare Function HidD_GetAttributes _
    Lib "hid.dll" _
    (ByVal HidDeviceObject As Long, _
    ByRef Attributes As HIDD_ATTRIBUTES) _
As Long
```

When an application calls the function, the function can change the values in the structure, and the application will see the new values.

Calling a Function

After the code has declared a function and any user-defined types it passes, the application may call the function.

Here is a call to the HidD_GetAttributes function declared above:

```
Dim Result as Long
Result = HidD_GetAttributes _
    (HidDevice, _
    DeviceAttributes)
```

HidDevice is a Long value returned by a previous API call. Result is non-zero on success. DeviceAttributes is a structure containing the Vendor ID, Product ID, and product version number retrieved from the device during enumeration.

Two Useful Routines

In addition to the basic API functions for USB communications, there are a couple of other API functions that I've found useful in HID and other applications. One copies data in memory, and the other returns text describing the last error detected by the operating system.

Moving Data in Memory

The API function RtlMoveMemory transfers a series of bytes from one location in memory to another. This function is useful for copying raw data between byte arrays and structures. This is the declaration:

```
Public Declare Function RtlMoveMemory _
    Lib "kernel32" _
    (dest As Any, _
    src As Any, _
    ByVal Count As Long) _
As Long
```

Rather than declaring the data address's (src) and destination (dest) as specific types, the values are declared As Any to allow flexibility in using the function. Count is the number of bytes to copy.

Here RtlMoveMemory copies four bytes from a structure into a byte array whose address will be passed in a call to the SetupDiGetDeviceInterfaceDetail function.

```
Call RtlMoveMemory _
    (DetailDataBuffer(0), _
    MyDeviceInterfaceDetailData, _
    4)
```

Viewing Errors

The second useful function is FormatMessage, which returns text describing the last error that Windows detected.

This is the function's declaration:

```
Public Declare Function FormatMessage _
    Lib "kernel32" _
    Alias "FormatMessageA" _
    (ByVal dwFlags As Long, _
    ByRef lpSource As Any, _
    ByVal dwMessageId As Long, _
    ByVal dwLanguageId As Long, _
    ByVal lpBuffer As String, _
    ByVal nSize As Long, _
    ByVal Arguments As Long) _
As Long
```

The function also uses the following system constant:

```
Public Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000
```

I use FormatMessage in a Visual-Basic function that returns the string containing the error message. During debugging, I call the function after making an API call and display the error, either in a list box or using a debug.print statement in the immediate window. This code is adapted from an example in Dan Appleman's *Win32 API Puzzle Book:*

```
Private Function GetErrorString _
    (ByVal LastError As Long) _
As String

    'Returns the error message for the last error.

    Dim Bytes As Long
```

```
Dim ErrorString As String
ErrorString = String$(129, 0)
Bytes = FormatMessage _
    (FORMAT_MESSAGE_FROM_SYSTEM, _
    0&, _
    LastError, _
    0, _
    ErrorString$, _
    128, _
    0)

'Subtract two characters from the message to
'strip the CR and LF.
If Bytes > 2 Then
    GetErrorString = Left$(ErrorString, Bytes - 2)
End If

End Function
```

Device Attachment and Removal

Other capabilities an application might want are detecting when a device is attached or removed from the bus and controlling whether or not an attached device is enabled. Windows provides ways to do this.

USBView

One way to search for a device is to search a list of every attached device. The Windows DDK includes C source code for the USBView application (Figure 15-1), which displays in tree form all hosts, hubs, and devices attached to the hubs. You can also view each device's descriptors. The code uses DeviceIoControl functions to retrieve the information. For a Visual-Basic application that does the same thing, I recommend the DisplayUSB example in John Hyde's book, *USB Design by Example*, which, by the way, is an excellent companion to this book.
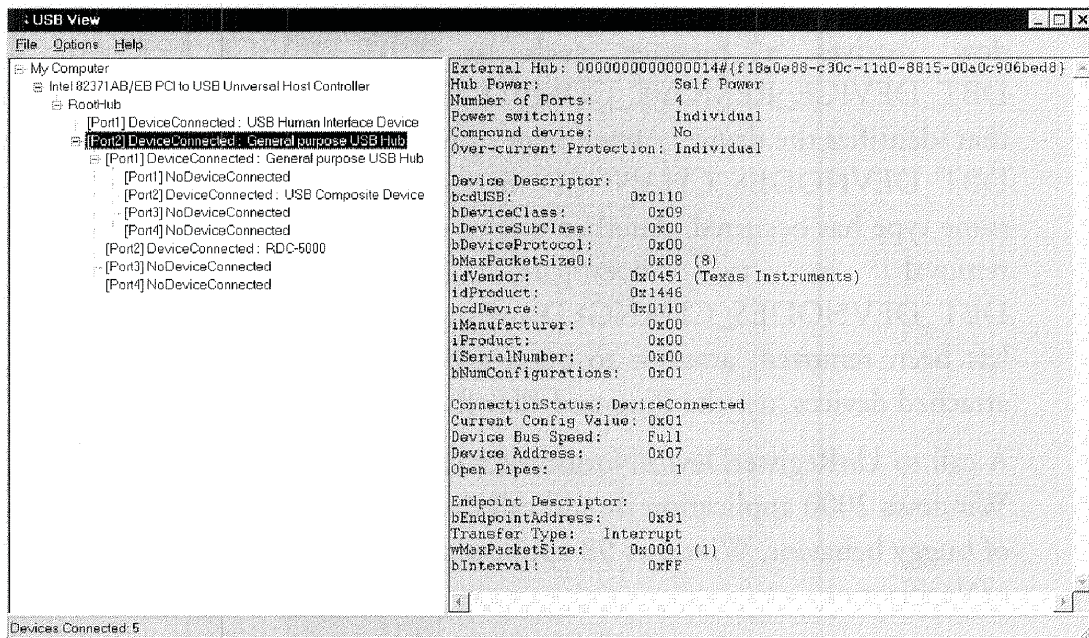
Figure 15-1: The USBView utility in the Windows DDK displays all hosts, hubs, and device attached to hubs.

Searching for a Device

To find out if a specific device is attached, an application can search using the Plug and Play/Device Management functions listed in Table 15-1 and described in greater detail in the next chapter. Searching can also reveal if a previously attached device has been removed. An application will also learn that a device is removed when it attempts to communicate and receives the error *invalid handle*.

Device Notification

Another way to learn of newly attached or removed devices uses Windows' RegisterDeviceNotification function. In calling the function, an application can pass a pointer to a structure containing the GUID of a device interface to monitor and a handle to a window to receive the event notifications.

USB Complete

When a device with a matching interface is attached or removed, the window receives a message such as DBT_DEVICE_ARRIVAL or DBT_DEVICE_REMOVE_COMPLETE with a pointer to a structure that identifies the device. Attachment or removal of a device also results in a DBT_DEVNODES_CHANGED message that indicates that an event of some type has occurred. Another way to detect a specific device's arrival or removal is to investigate further on receiving a DBT_DEVNODES_CHANGED message. To find out whether a device has been removed, attempt to open a handle to it. To search for newly attached devices, use the Plug-and-Play functions.

A call to UnRegisterDeviceNotification causes the notifications to cease. A Windows 2000 application should call this function before closing. Because of buggy behavior, Windows 98 applications shouldn't use UnRegisterDeviceNotification.

Enabling and Disabling Devices

The Windows 2000 DDK documents Setup functions that can enable or disable a device in software.

The CM_Request_Device_Eject function prepares a device for safe removal and physically ejects media that are ejectable. The SetupDiChangeState function can disable a device or load drivers for and start a device.

16

Human Interface Devices: Host Application Example

With the previous chapters' information about reports and how to call API functions, we're now ready to communicate with a HID. In this chapter, I present code that applications can use to communicate with HID-class devices. The examples are in both Visual-Basic and Visual C++. Headings identify text that is specific to a language. Much of the information applies to communications with any USB device.

Finding a Device

The first task is to find the device you want to communicate with. This involves examining properties of the HIDs available on a system and looking for a match, either in Vendor and Product IDs or in device capabilities. A series of API calls will accomplish this. The process uses many of the same Setup functions you would use to locate other USB devices.

Obtain the GUID for the HID Class

Before an application can communicate with a HID, it must obtain the globally unique identifier (GUID) for the HID class. Chapter 10 introduced the GUID, which is a 128-bit value that uniquely identifies an object. In this case, the object is the HID class. The GUID value is included in the file *hidclass.h*, so in theory you could hard-code it into the application. But you can also obtain the GUID by using an API function that reads the value from the system. Doing it this way, you'll be sure to have the correct value in the expected format.

The API call to retrieve the GUID for the HID class is HidD_GetHidGuid. The application doesn't have to do anything with the GUID itself. It just passes the GUID's address to other API functions.

Visual C++

This is the function's declaration:

```
VOID
  HidD_GetHidGuid(
    OUT LPGUID HidGuid
    );
```

This is the code to call the function:

```
HidD_GetHidGuid(&HidGuid);
```

Visual Basic

This is the function's declaration:

```
Public Declare Sub HidD_GetHidGuid _
    Lib "hid.dll" _
```

```
(ByRef HidGuid As GUID)
```

This routine has no return value, so it can be declared as a subroutine, as above. Or you can declare it as a function, with a return value of type Long, and ignore the returned value:

```
Public Declare Function HidD_GetHidGuid _
    Lib "hid.dll" _
    (ByRef HidGuid As GUID)
as Long
```

The GUID is returned in the variable HidGuid, which has the following user-defined type:

```
Public Type GUID
    Data1 As Long
    Data2 As Integer
    Data3 As Integer
    Data4(7) As Byte
End Type
```

HidGuid is declared byRef because Visual Basic requires user-defined types to be passed byRef.

The call to get the GUID is:

```
Call HidD_GetHidGuid(HidGuid)
```

or

```
Dim Result as Long
Result = HidD_GetHidGuid(HidGuid)
```

Get an Array of Structures with Information about the HIDs

The GUID enables the application to get information about a system's HIDs. The functions to do this are Windows Device Management Functions. There are two sets of essentially identical documentation for these in the Windows DDK documentation and in the Platform SDK in the MSDN documentation.

The SetupDiGetClassDevs function returns the address of an array of structures containing information about all attached and enumerated HIDs.

Visual C++

This is the function's declaration:

```
HDEVINFO
   SetupDiGetClassDevs(
      IN LPGUID  ClassGuid,  OPTIONAL
      IN PCTSTR  Enumerator,  OPTIONAL
      IN HWND  hwndParent,  OPTIONAL
      IN DWORD  Flags
      );
```

This is the code to call the function:

```
hDevInfo=SetupDiGetClassDevs
   (&HidGuid,
   NULL,
   NULL,
   DIGCF_PRESENT|DIGCF_INTERFACEDEVICE);
```

Visual Basic

This is the function's declaration:

```
Public Declare Function SetupDiGetClassDevs _
    Lib "setupapi.dll" _
    Alias "SetupDiGetClassDevsA" _
    (ByRef ClassGuid As GUID, _
    ByVal Enumerator As String, _
    ByVal hwndParent As Long, _
    ByVal Flags As Long) _
As Long
```

This is the code to call the function:

```
Public Const DIGCF_PRESENT = &H2
Public Const DIGCF_DEVICEINTERFACE = &H10

hDevInfo = SetupDiGetClassDevs _
    (HidGuid, _
    vbNullString, _
    0, _
    (DIGCF_PRESENT Or DIGCF_DEVICEINTERFACE))
```

Details

ClassGuid is HidGuid, the value returned in the last call. Enumerator and hwndParent are unused. The flags are two system constants defined in the file *setupapi.h*.

The flags tell the function to look only for device interfaces that are currently present (attached and enumerated) and that are members of the HID class, as specified in the ClassGuid parameter.

The value returned, hDevInfo, is the address of an array of structures containing information about all attached and enumerated HIDs. Again, there's no need to access the individual elements in the collection. You need the value only so you can pass it on in the next API call.

When the application is finished using the array pointed to by hDevInfo, it should free the resources used by calling the API function SetupDiDestroy-DeviceInfoList, as described later in this chapter.

Identify Each HID Interface

The next call is to SetupDiEnumDeviceInterfaces, which retrieves a pointer to a structure that identifies an interface in the previously retrieved Device-InfoSet array. Each call must specify one interface by passing an array index. To retrieve information about all of the interfaces, an application can use a loop to step through the array, incrementing the array index until the function returns zero, indicating that there are no more interfaces. The GetLastError API call will then return *No more data is available*.

How do you know if an interface is the one you're looking for? You don't, yet. The application needs more information before it can decide if it wants to use an interface. If the function returns multiple interfaces, the application will need to investigate each in turn, until it either finds what it's looking for or determines that the desired interface isn't present.

Again, the use for any returned pointers is to pass them on to the next function so we can learn more about the interfaces.

Visual C++

This is the function's declaration:

```
BOOLEAN
   SetupDiEnumDeviceInterfaces(
     IN HDEVINFO  DeviceInfoSet,
     IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
     IN LPGUID  InterfaceClassGuid,
     IN DWORD  MemberIndex,
     OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
     );
```

This is the declaration for DeviceInterfaceData's type:

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
    DWORD cbSize;
    GUID InterfaceClassGuid;
    DWORD Flags;
    ULONG_PTR Reserved;
} SP_DEVICE_INTERFACE_DATA,
*PSP_DEVICE_INTERFACE_DATA;
```

And this is the code to call the function:

```
devInfoData.cbSize = sizeof(devInfoData);
Result=SetupDiEnumDeviceInterfaces
  (hDevInfo,
  0,
  &HidGuid,
  MemberIndex,
  &devInfoData);
```

Visual Basic

This is the function's declaration:

```
Public Declare Function SetupDiEnumDeviceInterfaces _
    Lib "setupapi.dll" _
    (ByVal DeviceInfoSet As Long, _
    ByVal DeviceInfoData As Long, _
    ByRef InterfaceClassGuid As GUID, _
    ByVal MemberIndex As Long, _
    ByRef DeviceInterfaceData _
        As SP_DEVICE_INTERFACE_DATA) _
As Long
```

DeviceInterfaceData is a user-defined type:

```
Public Type SP_DEVICE_INTERFACE_DATA
    cbSize As Long
    InterfaceClassGuid As GUID
    Flags As Long
    Reserved As Long
End Type
```

This is the code to call the function:

```
Dim Result as Long
Dim MemberIndex as Long
Dim MyDeviceInterfaceData As SP_DEVICE_INTERFACE_DATA
'Store the size of the structure
MyDeviceInterfaceData.cbSize = _
   LenB(MyDeviceInterfaceData)
Result = SetupDiEnumDeviceInterfaces _
    (DeviceInfoSet, _
    0, _
    HidGuid, _
    MemberIndex, _
    MyDeviceInterfaceData)
```

Details

The parameter cbSize is the size of the SP_DEVICE_INTERFACE_DATA structure in bytes. Before calling SetupDiEnumDeviceInterfaces, the size must be stored in the structure that the function will pass. Use the `sizeof` operator in Visual C++ or the `LenB` operator in Visual Basic to retrieve the size, which is 28 bytes: 4 for each Long and 16 for the GUID, which contains one Long (4 bytes), two Integers (4 bytes), and eight Bytes. The other values in the structure should be zero.

Two of the values passed to this function are values returned previously: HidGuid and DeviceInfoSet. DeviceInfoData is an optional pointer to an SP_DEVINFO_DATA structure that limits the search to interfaces of a particular device. MemberIndex is the index of the DeviceInfoSet array. MyDeviceInterfaceData is the returned structure that identifies an interface of the requested type, which in this case is a HID.

Get the Device Pathname

The next API call, SetupDiGetDeviceInterfaceDetail, returns yet another structure. This time the structure relates to a device interface identified in the previous call. The structure's DevicePath member is a device pathname that the application can use to open communications with the device.

Before calling this function for the first time, there's no way to know the value of DeviceInterfaceDetailDataSize, which must contain the size in bytes of the DeviceInterfaceDetailData structure. Yet the call won't return the structure unless it has this information. The solution is to call the function twice. The first time, GetLastError will return the error *The data area passed to a system call is too small,* but the RequiredSize parameter will contain the correct value for DeviceInterfaceDetailDataSize. The second time, you pass the returned value and the function succeeds.

Visual C++

This is the function's declaration:

```
BOOLEAN
   SetupDiGetDeviceInterfaceDetail(
      IN HDEVINFO  DeviceInfoSet,
      IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
      OUT PSP_DEVICE_INTERFACE_DETAIL_DATA
         DeviceInterfaceDetailData,  OPTIONAL
      IN DWORD  DeviceInterfaceDetailDataSize,
      OUT PDWORD  RequiredSize,  OPTIONAL
      OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
      );
```

This is the declaration for DeviceInterfaceDetailData's structure:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
    DWORD cbSize;
    TCHAR DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA,
*PSP_DEVICE_INTERFACE_DETAIL_DATA;
```

This is the code to call the function twice, first to get the structure's size, and second to get a pointer to the structure:

```
// Get the Length value.
```

```
// The call will return with a "buffer too small"
// error which can be ignored.
Result = SetupDiGetDeviceInterfaceDetail
  (hDevInfo,
  &devInfoData,
  NULL,
  0,
  &Length,
  NULL);

// Allocate memory for the hDevInfo structure,
// using the returned Length.
detailData =
  (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(Length);

// Set cbSize in the detailData structure.
detailData -> cbSize =
  sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Call the function again, this time passing it the
// returned buffer size.
Result = SetupDiGetDeviceInterfaceDetail
  (hDevInfo,
  &devInfoData,
  detailData,
  Length,
  &Required,
  NULL);
```

Visual Basic

The function's declaration is:

```
Public Declare Function _
    SetupDiGetDeviceInterfaceDetail _
    Lib "setupapi.dll" _
    Alias "SetupDiGetDeviceInterfaceDetailA" _
    (ByVal DeviceInfoSet As Long, _
    ByRef DeviceInterfaceData _
        As SP_DEVICE_INTERFACE_DATA, _
    ByVal DeviceInterfaceDetailData As Long, _
    ByVal DeviceInterfaceDetailDataSize As Long, _
    ByRef RequiredSize As Long, _
    ByVal DeviceInfoData As Long) _
```

```
As Long
```

The structure returned in DeviceInterfaceDetailData is a user-defined type:

```
Public Type SP_DEVICE_INTERFACE_DETAIL_DATA
    cbSize As Long
    DevicePath As String
End Type
```

Because of the different string formats used by Visual Basic and C, you can't pass this structure in the usual way, using ByRef to pass the structure's address. But there is a way around the problem. The first step is to allocate a buffer in memory to hold the structure. Then you can use the VarPtr operator to get the starting address of the buffer, and pass the address ByVal. When the function returns, you can copy the data in the buffer into a DeviceInterfaceDetailData structure, or just extract the data of interest, which is the device pathname.

This is the code for the first call:

```
Dim Needed as Long
Result = SetupDiGetDeviceInterfaceDetail _
    (DeviceInfoSet, _
    MyDeviceInterfaceData, _
    0, _
    0, _
    Needed, _
    0)
```

DeviceInfoSet and MyDeviceInterfaceData are structures returned by previous calls. After calling this function, Needed contains the buffer size to pass in the next call.

Before calling the function again, we need to take care of a few things.

The DetailData variable to be passed in the next call is set to equal the value returned in Needed:

```
Dim DetailData as Long
DetailData = Needed
Dim DetailDataBuffer() as Byte
```

The size of the structure to be returned is stored in its cbSize parameter:

```
'Store the structure's size.
```

```
MyDeviceInterfaceDetailData.cbSize = _
Len(MyDeviceInterfaceDetailData)
```

Because we're going to pass only the address of a byte array for the returned structure, we need to allocate enough memory in the array to hold the structure:

```
ReDim DetailDataBuffer(Needed)
```

The first four bytes of the byte array hold the array's size, which can be copied from the cbSize property in the MyDeviceInterfaceDetailData structure:

```
Call RtlMoveMemory _
    (DetailDataBuffer(0), _
    MyDeviceInterfaceDetailData, _
    4)
```

Now we're ready to call SetupDiGetDeviceInterfaceDetail again:

```
'Call SetupDiGetDeviceInterfaceDetail again.
'This time, pass the address
'of the first element of DetailDataBuffer
'and the returned required buffer size in DetailData.
Result = SetupDiGetDeviceInterfaceDetail _
            (DeviceInfoSet, _
            MyDeviceInterfaceData, _
            VarPtr(DetailDataBuffer(0)), _
            DetailData, _
            Needed, _
            0)
```

VarPtr(DetailDataBuffer(0)) is the starting address of the byte array that will contain the MyDeviceInterfaceDetailData structure. DetailData holds the size returned by the previous call.

The item of interest in the returned structure is the device pathname to be used in additional API calls. To extract the pathname from the byte array, convert the byte array to a string, convert the result to Unicode for compatibility with Visual Basic, and strip the cbSize characters from the beginning of the string.

```
'Convert the byte array to a string.
DevicePathName = CStr(DetailDataBuffer())
'Convert to Unicode.
DevicePathName = StrConv(DevicePathName, vbUnicode)
```

```
'Strip cbSize (4 characters) from the beginning.
DevicePathName = _
    Right$(DevicePathName, Len(DevicePathName) - 4)
```

Get a Handle for the Device

Now that we have a device pathname, we're ready to open communications with the device itself. The first step is the all-purpose function CreateFile, which can open a handle to a file or any device whose driver supports CreateFile. Devices with HID interfaces are among these.

On success, the value returned by CreateFile is a handle that other API functions can use to exchange data with the device.

Visual C++

This is the function's declaration:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

This is the code to call the function:

```
DeviceHandle=CreateFile
  (detailData->DevicePath,
  GENERIC_READ|GENERIC_WRITE,
  FILE_SHARE_READ|FILE_SHARE_WRITE,
  NULL,
  OPEN_EXISTING,
  0,
  NULL);
```

Visual Basic

This is the function's declaration:

```
Public Declare Function CreateFile _
    Lib "kernel32" _
```

```
        Alias "CreateFileA" _
        (ByVal lpFileName As String, _
        ByVal dwDesiredAccess As Long, _
        ByVal dwShareMode As Long, _
        ByVal lpSecurityAttributes As Long, _
        ByVal dwCreationDisposition As Long, _
        ByVal dwFlagsAndAttributes As Long, _
        ByVal hTemplateFile As Long) _
    As Long
```

And this is the code to call the function:

```
Dim HidDevice As Long
HidDevice = CreateFile _
    (DevicePathName, _
    GENERIC_READ Or GENERIC_WRITE, _
    (FILE_SHARE_READ Or FILE_SHARE_WRITE), _
    0, _
    OPEN_EXISTING, _
    0, _
    0)
```

The function passes a pointer to the DevicePathName string returned in the previous call. The parameter is declared as a String to be passed ByVal, because of Visual Basic's different string format, as explained earlier.

The constants passed by the call are defined in several locations, including *winnt.h* and *wdm.h*, and must be declared in a declarations section of a module in the Visual-Basic application:

```
Public Const GENERIC_READ = &H80000000
Public Const GENERIC_WRITE = &H40000000
Public Const FILE_SHARE_READ = &H1
Public Const FILE_SHARE_WRITE = &H2
Public Const OPEN_EXISTING = 3
```

Details

When the application no longer needs to access the device, it should free system resources by calling the CloseHandle API function, as described later in this chapter.

Read the Vendor and Product IDs

One way to identify whether or not a device is the one you want is to get its Vendor and Product IDs and compare them with the IDs for the product you're looking for. This is the way to find custom devices that don't fit standard usages. For other devices, this information may not be important, and if not, you can skip this step.

The API function HidD_GetAttributes retrieves a pointer to a structure containing the Vendor and Product IDs and the product's version number.

Visual C++

This is the function's declaration:

```
BOOLEAN
  HidD_GetAttributes(
    IN HANDLE HidDeviceObject,
    OUT PHIDD_ATTRIBUTES Attributes
    );
```

The HIDD_ATTRIBUTES structure contains the information about the device:

```
typedef struct _HIDD_ATTRIBUTES {
  ULONG    Size;
  USHORT   VendorID;
  USHORT   ProductID;
  USHORT   VersionNumber;
} HIDD_ATTRIBUTES, *PHIDD_ATTRIBUTES;
```

This is the code to retrieve the structure:

```
// Set the Size member to the number of bytes
// in the structure.
Attributes.Size = sizeof(Attributes);
Result = HidD_GetAttributes
  (DeviceHandle,
  &Attributes);
```

Visual Basic

This is the declaration for the function:

```
Public Declare Function HidD_GetAttributes _
```

```
      Lib "hid.dll" _
      (ByVal HidDeviceObject As Long, _
      ByRef Attributes As HIDD_ATTRIBUTES) _
As Long
```

The HIDD_ATTRIBUTES structure contains the information about the device:

```
Public Type HIDD_ATTRIBUTES
    Size As Long
    VendorID As Integer
    ProductID As Integer
    VersionNumber As Integer
End Type
```

This is the code to retrieve the structure:

```
Dim DeviceAttributes As HIDD_ATTRIBUTES
'Set the Size property to the number of bytes
'in the structure.
DeviceAttributes.Size = LenB(DeviceAttributes)
Result = HidD_GetAttributes _
    (HidDevice, _
    DeviceAttributes)
```

Details

The HidDeviceObject parameter is the handle returned by CreateFile. If the function returns a non-zero value, the DeviceAttributes structure filled without error.

The application can then compare the retrieved values with the desired Vendor and Product IDs and version number.

If it isn't a match, the application should use the CloseHandle API call to close the handle to the interface. The application can then move on to test the next HID detected by SetupDiEnumDeviceInterfaces. When the application is finished examining the HIDs, it should free the resources reserved by SetupDiGetClassDevs by calling SetupDiDestroyDeviceInfoList.

Get a Pointer to a Buffer with Device Capabilities

Another way to find out more about a device is to examine its capabilities. You can do this for a device whose Vendor and Product IDs matched the values you were looking for, or you can examine the capabilities for an unknown device.

The first task is to get a pointer to a buffer with information about the device's capabilities. The API call to do this is HidD_GetPreparsedData.

Visual C++

This is the function's declaration:

```
BOOLEAN
   HidD_GetPreparsedData(
      IN HANDLE HidDeviceObject,
      OUT PHIDP_PREPARSED_DATA *PreparsedData
      );
```

This is the code to call the function:

```
PHIDP_PREPARSED_DATA PreparsedData;
HidD_GetPreparsedData
   (DeviceHandle,
   &PreparsedData);
```

Visual Basic

This is the function's declaration:

```
Public Declare Function HidD_GetPreparsedData _
   Lib "hid.dll" _
   (ByVal HidDeviceObject As Long, _
   ByRef PreparsedData As Long) _
As Long
```

This is the code to call the function:

```
Result = HidD_GetPreparsedData _
   (HidDevice, _
   PreparsedData)
```

HidDeviceObject is the handle returned by CreateFile. PreparsedData is a pointer to the buffer containing the data. The application doesn't need to

access the data in the buffer; it just needs to pass its starting address to another API function.

When the application no longer needs to access the PreparsedData, it should free system resources by calling HidD_FreePreparsedData, as described later in this chapter.

Get the Device's Capabilities

The HidP_GetCaps function returns a structure that contains information about the device's capabilities. The structure contains the device's Usage, Usage Page, report lengths, and the number of button capabilities, value capabilities, and data indices for Input, Output, and Feature reports, as stored in the device's firmware. If you didn't use the Vendor and Product IDs to identify the device, the capabilities information can help you decide if you want to continue communicating with the device. Even if you know that you have the device you're looking for, the report lengths and other information are useful in determining what kinds of data you can transfer. Not every item in the structure applies to all devices.

Visual C++

This is the function's declaration:

```
NTSTATUS
    HidP_GetCaps(
        IN PHIDP_PREPARSED_DATA PreparsedData,
        OUT PHIDP_CAPS Capabilities
        );
```

This is the declaration for the HIDP_CAPS structure:

```
typedef struct _HIDP_CAPS {
  USAGE Usage;
  USAGE UsagePage ;
  USHORT InputReportByteLength ;
  USHORT OutputReportByteLength ;
  USHORT FeatureReportByteLength ;
  .

  .

  USHORT NumberLinkCollectionNodes ;
  USHORT NumberInputButtonCaps ;
```