By Stuart Allman, Cypress Semiconductor

THE USB HID CLASS IS A POWERFUL AND VERSATILE WAY TO GET YOUR DEVICE ON THE USB. IF YOUR USB DEVICE CAN EXIST WITHIN THE BANDWIDTH LIMITS OF THE HID DRIVER, THEN USING THIS DRIVER MAY SAVE YOUR SANITY AND YOUR SCHEDULE. AN EXAMPLE SHOWS HOW.

Using the HID class eases the job of writing USB device drivers

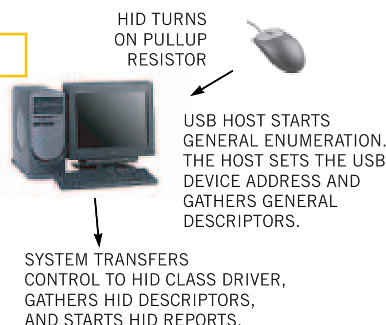
THE DIFFICULTY OF WRITING device drivers is one of the major barriers to the adoption of the USB. A typical embedded-device engineer who can comfortably design embedded systems all day long can get nervous at the thought of writing a PC-device driver. The design is no longer as simple as using the parallel port or serial port. Nonetheless, the USB offers many advantages, including multiplatform support, standard device classes, and support for vendor-defined devices. Motherboard vendors are also attempting to eliminate the legacy connections in favor of the USB, which forces embedded-system engineers to accept the evolution of technology. Move to the USB, or become obsolete.

Fortunately, by using the HID (Human Interface Device) class for the USB, you need not write a single Windows/Mac/Linux device driver. Even if your device doesn't fit into one of the predefined HID usages, you can use a completely vendor-defined usage.

The HID class is a standard device classification for the USB, but don't let the words "standard" or "classification" fool you. The HID class doesn't represent a set of fixed-function devices. Rather, it supports a variety of devices with widely varying characteristics. Some examples include a computer mouse, a keyboard, sports equipment, medical instruments, audio/video devices, and vendor-defined functions. HID also supports device primitives such as an LED and a button, and standard measurements, such as time, temperature, and distance. The general idea is that you can use the HID class to support a range of vendor-specific applications that a USB-device class doesn't currently support. Microsoft also supports the HID class with a well-rounded library that it includes as part of the Windows DDK.

To show how to use the HID class to communicate with an embedded device, a thermometer example that follows focuses on encapsulating the concepts of a Windows-based application. This example also shows how to use the Windows DDK library in a typical situation for a vendor-defined HID usage. The goal of the sample Windows application and USB device is to show the basic steps of data ex-

Figure 1



The HID device-enumeration process begins with the action of a simple pull-up resistor.

change across the USB using the HID driver. You can find a complete design package for this example with the Web version of this story at www.edn.com.

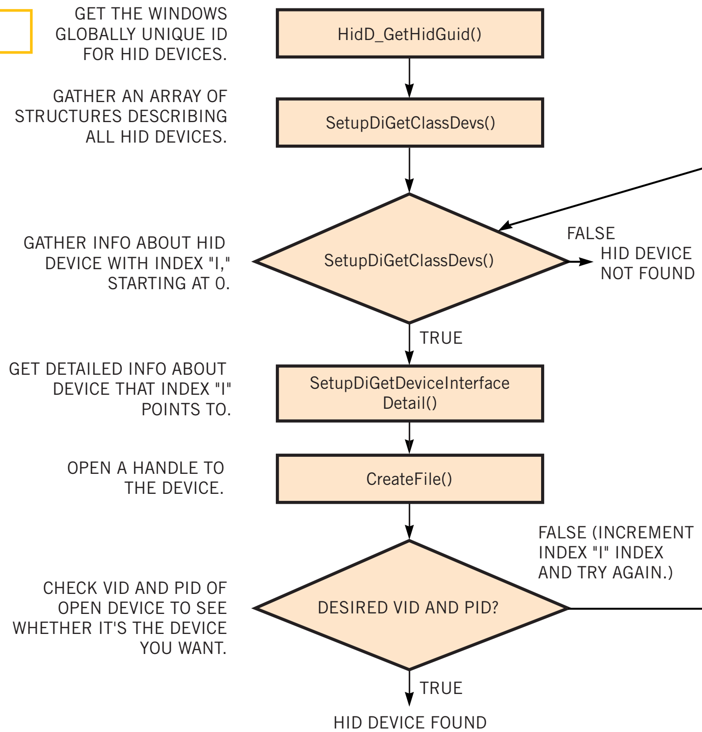
HOW DOES A DEVICE GET ON THE USB?

The process of a device's getting on the USB begins with the act of a simple pull-up resistor. After you plug the device into an active USB host, the device turns on a pull-up resistor on one of the USB data signals. The device pulls up D− if the device is low-speed and pulls up D+ if it is full- or high-speed. The USB host detects this condition and begins device *enumeration*, a start-up process on the USB (**Figure 1**). (Note that high-speed hosts go through an additional process.) During enumeration, the host requests a number of data structures, or *descriptors*, from the device. These descriptors contain information about the number and type of communication channels, or *endpoints*, that the USB device desires to use, as well as information about any device class. Enumeration occurs on the default endpoint, which is endpoint 0, also known as the control endpoint. The host also assigns a unique 7-bit address to the device, directing communications to a particular device.

On PC or Mac platforms, the first part of the enumeration process occurs without custom or class-specific device drivers. During the first part of enumeration, the USB driver retrieves the general descriptors that all devices support and gives the device a unique address on the USB. If the descriptors indicate that the device belongs to a particular USB device class, the USB driver hands off the rest of the enumeration to the specified class driver. In the example for this article, the driver handoff is to the HID-class driver. If the descriptors don't specify a class, then the device's VID (vendor ID) and PID (product ID), gathered from the descriptors, identify an appropriate driver for the USB device.

In a properly designed USB HID device, you need not worry about the driver process. The device drivers on the USB host handle the enumeration process and the class-driver handoff for you. The USB host simply enumerates the device, which is then ready to use with your application. However, if you improperly format the descriptors or the device responds improperly, either the USB or the class driver disables the device, depending on when the failure occurred during enumeration.

Figure 2



Opening a device takes seven steps.

HID REQUIRES NO HUMAN INTERFACE

Don't let the "human interface" part of HID confuse you. A USB device that claims to belong to the Human Interface Device class need not necessarily have a human interface. The HID class supports a variety of devices that you would generally never associate with a human interface. Devices can be as obvious as a mouse or a keyboard but as unexpected as an IC programmer or a thermometer. The class also supports a variety of device primitives that you would associate with a general digital device, such as an LED, a button, a vector, or an LCD.

The HID class allows you to create a collection of HID usages to describe a device. The HID report descriptor, which the host gathers during enumeration, describes this collection along with the input and output data streams. For instance, a clock description could be an input stream of 3 bytes corresponding to hours, minutes, and seconds. The clock could also have an output report in the collection that describes how to write to a device's LCD. The HID class also has the capacity to allow vendor-defined usages. Even if none of the predefined usages in the HID usage tables (**Reference 1**) can describe your device, you can still reserve a generic vendor-defined slot in the HID report collection.

The downside to using the HID class is that the typical class drivers support only one type of endpoint communication—an interrupt—and the bandwidth is limited to well below the USB's maximum throughput. The bandwidth is limited to 800 bytes/sec/report (8 bytes/msec/I/O report) for low-speed devices, 64,000 bytes/sec/report (64 bytes/msec/I/O report) for full-speed devices, and approximately 23.4 Mbytes/sec/report (3072 bytes/microframe/I/O report) for high-speed devices. An HID may specify multiple reports if more than the "per-report" bandwidth is necessary.

Don't let these bandwidth limits immediately discourage you. Using an intelligent USB controller often helps presort data and lower the bandwidth that's required to support an application. The

low-speed bandwidth limit was far greater than was necessary for the sample application that follows.

THE STEPS TO A WORKING HID

You need to take many steps to make an HID enumerate and communicate using HID reports on the USB. This process isn't easy, but this application provides a framework that you can copy and edit as a starting point.

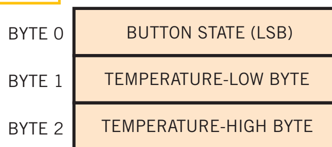
All USB devices handle "Chapter 9 requests," which Chapter 9 of the USB specification describes (**Reference 2**). The requests include tasks such retrieving a standard set of descriptors from the device and setting standard device parameters. The first specific task in making an HID is specifying the HID class in the USB interface descriptor's bInterfaceClass field. A value of 0x03 corresponds to the HID class. This value lets the general USB driver know which class driver the application needs to be passed on to after enumeration is complete.

During enumeration, the device also supplies an HID class descriptor as a subsection of the configuration descriptor in between the interface and the endpoint-descriptor subsections. This descriptor basically contains the HID-specification version information and the length of the HID-report descriptor. The second aspect of making an HID is the HID-report descriptor itself, which can be complex. The HID descriptor comprises HID usages, each of which describes a field in an input or an output report (**Listing 1**). The HID usage-table document shows a large number of useful examples of how to create an HID-report descriptor (**Reference 1**).

When the device correctly handles the Chapter 9 requests and HID-specific parameters, the remainder of the work lies in the application itself. This example uses a Microsoft Windows-based application. Examples are also available elsewhere for MacOS and Linux (**references 3** and **4**).

An application must first be able to open the device for communications—a fairly lengthy process. However, the sample application encapsulates the process into a reusable function, bOpenHidDevice() (**Reference 5**). The bOpenHidDevice function takes a device-handle reference, a target device VID, and a target

Figure 3



The HID input report for the thermometer includes the button state and temperature measurement in one report.

device PID as function arguments. If the host finds a device that matches the desired VID and PID, then the function returns "true," and the Windows HID API assigns a valid value to the device handle. If the function returns false, it more than likely means that the either the device is not plugged in or the device failed enumeration.

The process of opening a device consists of seven steps (**Figure 2**).

1. Obtain the Windows GUID (globally unique ID) for HID devices via a call to HidD_GetHidGuid().

2. Get an array of structures that contain information about all attached HIDs via a call to SetupDiGetClassDevs(). This step uses the previously obtained HID GUID to specify that the list should contain only HID devices.

3. Use the Windows function SetupDiEnumDeviceInterfaces() to get information about a device in the list. You need to step through each index of device information until you find one with the correct VID and PID. If this function returns "false," then you have reached the end of the list without finding the desired device.

4. A call to SetupDiGetDeviceInterfaceDetail() returns detailed data about the device indexed in the previous step. You want to use the device path to open the device in the next step.

5. Call CreateFile() to open the device using the path obtained in the previous step. If the Windows API call to CreateFile() returns a valid handle, then you can examine the VID and PID to determine whether this is the device you want.

6. Compare the open device's VID and PID to determine whether this is the device you want. If so, then you should return the device handle and a "true" condition.

7. If the VID and PID are incorrect, then you need to close the device handle and return to Step 3 to check the next device the list indexes.

LISTING 1–HID-REPORT DESCRIPTOR

```
Usage Page (vendor defined)
Usage (vendor defined)
Collection (application)
Usage (vendor defined)

        ** input report
        Usage (vendor defined)
        Logical Minimum (-128)
        Logical Maximum (127)
        Report Count (1)
        Report Size (8)
        Input (Data)

        ** output report
        Usage (vendor defined)
        Logical Minimum (-128)
        Logical Maximum (127)
        Report Count (1)
        Report Size (8)
        Output (Data)

End Collection
```

The HID application must then handle device-attachment and -detachment notification. The bHidDeviceNotify() function encapsulates this process. This function causes the Windows USB system to send a WM_DEVICECHANGE message to the application whenever a USB HID device is plugged into or unplugged from the system. The notification system is not intelligent enough to indicate an HID-device change for a VID and a PID, so the application must check whether the device is still attached on any notification message.

Setting up device notifications is a five-part process:

1. Obtain the Windows GUID for HID devices by way of a call to HidD_Get HidGuid().

2. Clear the contents of a DEV_ BROADCAST_DEVICEINTERFACE structure 0.

3. Assign the members of the structure such that you specify the HID GUID.

4. Register the application for device notifications by calling the function RegisterDeviceNotification().

5. If the previous step returns an in-

valid handle, then an error has occurred, and the function should return an error. Otherwise, the return is successful.

The Windows DDK provides a number of functions that allow you to find devices with certain capabilities and retrieve pieces of data from the USB device. The library functions are too numerous to mention here, but you can find them by searching for "HID Support Routines for Clients" on the MSDN Web site (**Reference 6**).

INPUT AND OUTPUT REPORTS

The next function that the application needs is the ability to read input reports and write output reports. To perform these tasks, an application uses the ReadFile() and WriteFile() functions, respectively. Each of these functions use the device handle that the CreateFile() function returns. This application uses overlapped transactions, which allow the application to enter a low-CPU-overhead state while waiting for a requested USB transaction to occur. You can queue as many as eight overlapped transactions at once by default, and you can queue more if more are specified. If an application doesn't use overlapped transactions, then it has to wait until the USB transaction finishes before it can perform any additional tasks. This state is undesirable when an application spends most of its time waiting for data returned from a USB device. The bOpenHidDevice() function opens the device with the FILE_FLAG_OVERLAPPED parameter, so all transactions are overlapped in this example.

A second aspect of data transfer is the use of multiple threads. Because the main application needs to remain responsive even while waiting for USB transactions, it is desirable to put the USB reads and writes in a separate thread. This feature allows the main application to continue processing messages while the USB transaction thread waits for overlapped transactions to complete.

A SAMPLE HID THERMOMETER

The thermometer in this example communicates with a PC via the HID-class driver. This example shows an unconventional use of the versatile HID class. This thermometer has an LED, a button, and a temperature measurement. The button and thermometer are grouped into an input report, and the host application controls the LED via an output report. Again, for a complete design package, see www.ednmag.com.

An LED is a predefined "usage page"—that is, a collection of LED-related functions—of the HID class (see the HID usage-tables specification). For the purposes of this application, the LED usage is defined as a ready LED. The intention of providing this control is to show how to use an HID output report. The report comprises one byte that turns on the LED (0x01) or off (0x00). The LED is on when the computer starts taking measurements and off when the measurements stop; that is, when the application closes. WriteFile() sends an overlapped HID output report to the USB device.

The button is also a predefined usage page of the HID class. You can define a general-purpose button without a usage in mind. **Figure 3** shows the format of the button report. To request button information from the USB device, the windows application uses an overlapped transaction via a call to ReadFile().

The temperature measurement is a bit more complex. The "generic-desktop" usage page contains a nonoriented vector. The intended use of a nonoriented vector is for measuring time, distance, temperature, light intensity, and so on when direction is not important. The HID report descriptor also indicates the units for the measurement—that is, SI linear Kelvin—and the exponent (10^0) for the reported value. The temperature-measurement IC on the USB device reports in Celsius, so reporting in Kelvin is a matter of simply adding 273 to the IC's measured value. The application searches for a USB device; it does not look for units, exponents, or other information. However, the Windows HID DDK allows your application to search for devices with certain capabilities instead of a dedicated device (**Reference 6**).

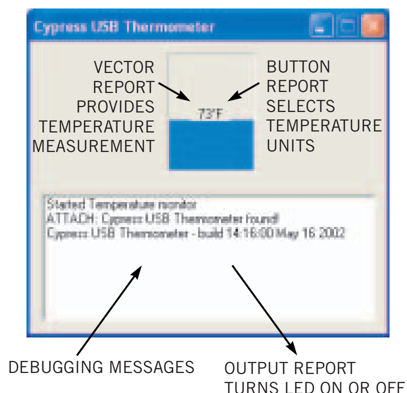


Figure 4

The HID thermometer application reports temperature measurement, debugging messages, and the output report on the screen.

LISTING 2–HID-REPORT DESCRIPTOR WITH THREE REPORTS

```
Usage Page (vendor defined)
Usage (vendor defined)
Collection (application)

        Usage Page (button)
        Usage Minimum (1)
        Usage Maximum (1)
        Logical Minimum (0)
        Logical Maximum (1)
        Report Count (1)
        Report Size (1)
        Input (Data)
        Report Count (1)
        Report Size (7)
        Input (Constant)

        Usage Page (Generic Desktop)
        Usage (non-oriented vector)
        Logical Minimum (218)
        Logical Maximum (393)
        Units (SI Linear Kelvin)
        Units Exponent (0)
        Report Count (1)
        Report Size (16)
        Input (Data)

        Usage Page (LED)
        Usage (Ready)
        Logical Minimum (0)
        Logical Maximum (1)
        Report Count (1)
        Report Size (1)
        Output (Data)
        Report Count (1)
        Report Size (7)
        Output (Constant)

End Collection
```

The example provides both the button and temperature measurement as one report. Whenever the button changes or a temperature measurement is complete, the USB device reports these values back to the host in the format that **Figure 3** shows. The format of the HID report descriptor for the thermometer shows the vendor-defined usage and a collection of three reports, two of which are in a single input report (**Listing 2**). The Windows HID driver follows the conventions set in the HID-class specification to parse the HID report descriptor. Even if the host-side application doesn't use some of the information in the HID report descriptor, it is still necessary for the HID driver to accept your device.

Detailed information about how to format HID report descriptors is beyond the scope of this article. The HID-class and HID usage-table specifications provide complete information on this subject. A HID descriptor tool is also available to help you with this task.

The host application for the HID-thermometer application was written using the VisualC++ Version 6.0 tools and the Win98 DDK (**Figure 4**). The Win98 DDK isn't the most up-to-date kit, but it ensures that you can build the application using all versions of the Windows DDK and that the application will run with all USB-enabled versions of Windows.

Cypress developed the USB device for the CY7C63743 USB microcontroller. The CY3644 application board that comes with the microcontroller's development kit has buttons, LEDs, and a temperature IC. The microcontroller's source code, assembler, and ROM image are part of the design package.□

References

1. USB HID specification, HID-usages tables, and HID descriptor tool, www.usb.org/developers/hidpage.html.
2. USB specification, www.usb.org/developers/docs.html.
3. Apple USB, http://developer.apple.com/hardware/usb/.
4. Linux USB, www.linux-usb.org.
5. Axelson, Jan, *USB Complete*, Lakeview Research, www.lvr.com.
6. Microsoft DDK, http://msdn.microsoft.com.
7. Hyde, John, *USB Design by Example*, Intel, www.usb-by-example.com.

Author's Biography

Stuart Allman is a senior systems engineer and solution architect for the Personal Communications Division of Cypress Semiconductor (www.cypress.com). He holds a BS degree in electrical engineering from the University of Washington (Seattle). Before joining Cypress, Allman worked at small audio companies developing microcontroller- and DSP-based products. Allman joined Cypress in 1998 and has been involved with microprocessor development tools, IC architectures, and encryption technology.