If you’re developing a device that will connect to PCs, chances are these days you’ll be using the Universal Serial Bus (USB). USB is designed to serve as a replacement for most of the PC’s legacy ports: parallel, serial, and so on.

USB is versatile enough for use with many common peripheral types as well as more specialized devices. Standard peripherals that can use USB include mice, keyboards, disk drives, printers, and scanners. USB is also becoming popular for data acquisition units, test instruments, and monitoring and control devices. Just about anything that in the past would have connected to an RS-232 or parallel port is a candidate for USB.

For many specialized devices, the human interface device (HID) class offers a quick path for USB developers. Although the HID specification was written to meet the needs of mice, keyboards, and similar input devices, it’s broad enough to be useful for other devices that need to exchange data in both directions at moderate rates.

Because Windows 98 and 2000 include HID-class drivers, there’s no need to write a device driver for your new gadget. And the firmware requirements to classify a device as HID are minimal, consisting mainly of a series of data structures that describe the HID interface and the data to be exchanged.

The main limitation of HID is speed, with a maximum transfer rate of 64KB/s. This is much less than the full-speed bus rate of 12Mbits/s, but still fast enough for plenty of applications.

This article is an introduction to HID development. I’ll describe the firmware requirements to enable the operating system to detect the HID and exchange information with it. I’ll also introduce the API functions available in Windows for communicating with HIDs. And I’ll show how to use a couple of

The inclusion of a HID-class USB driver in Windows 98 and 2000 makes connecting your embedded device to a PC easier than ever. Here’s everything you need to know to create a Human Interface Device.
free tools that will assist you in developing the firmware and ensuring that Windows can communicate with the device.

For a primer on USB in general, see Jack Ganssle’s introduction to the topic, “An Introduction to USB Development” (March 2000, p. 79).

**What is a HID?**

The HID-class specification is the product of a working group sponsored by the USB Implementers Forum ([www.usb.org](http://www.usb.org)). The forum is the organization formed by the companies responsible for developing the USB specification.

The idea behind classes is to make development easier by defining requirements and behaviors shared by devices with similar functions. An operating system can include a device driver based on a class specification, and a device that conforms to the specification can use the class driver instead of having to provide its own special-purpose driver. This is a big time saver.

The HID class was one of the first USB classes to be supported under Windows. No doubt this was because the class includes essential peripherals like keyboards and pointing devices, which were intended from the outset to use USB. The class specification and additional documents and tools are available from the USB Implementers Forum’s Web site.

The term human interface suggests that these devices interact directly with humans. With a keyboard or mouse, a human’s actions are indeed what determines the data that will be sent to the host. Other examples of HIDs include front panels, remote controls, telephone keypads, and game controls. But a HID doesn’t have to have button, knobs, or switches. The specification mentions bar-code readers, thermometers, and voltmeters as other devices that could fall into the HID class.

HIDs can also receive data from the host. With a force-feedback joystick,
users can experience effects that match their actions, such as more resistance when pulling the stick to cause a simulated airplane to climb. Other possible HIDs are remote displays, robots, and devices controlled by a virtual control panel on the host computer.

In short, a HID can be any device that can function within the limits defined by the specification. The major features and limitations are:

- A full-speed HID can transfer up to 64,000 bytes per second (64 bytes in each 1ms frame). A low-speed device is guaranteed only 800 bytes per second (eight bytes every 10ms)
- A HID can request the host to poll the device periodically to find out if the device has data to send
- All data exchanged by a HID resides in defined data structures called reports. A single report can contain up to 65,535 bytes. The device’s firmware must include a report descriptor that describes the data to be exchanged. The report format is flexible enough to handle just about any type of data

**Device requirements**

In many ways, HIDs are no different from other USB devices. Every USB peripheral must contain an intelligent controller that knows how to respond to requests and other events at its USB port.

The controller must have one or more endpoints, which are buffers that store received data or data waiting to be sent. Each endpoint has a number and direction. Endpoint 0 is bidirectional and is required for the control transfers used in enumerating the device.

Every device must respond to a series of control requests defined by the specification. For example, when the host sends a Get_Status request, the device must return status information in the expected format. Every device must also store the descriptors that the host will request.

The device’s serial interface engine (SIE) handles the details of sending and receiving the individual bits on the bus, much as a UART does for asynchronous serial communications. Beyond this, the amount of firmware support required to complete a transfer varies with the transfer type and the chip architecture.

Besides the requirements that apply to all devices, HIDs have additional requirements:

- A HID must have an interrupt IN endpoint for sending periodic data to the host. An interrupt OUT endpoint for receiving periodic data from the host is optional
- A HID must contain a class descriptor and one or more report descriptors
- A HID must support the HID-specific control request Get_Report and may support the optional request Set_Report
- For interrupt IN transfers, the device must place the report data in the interrupt endpoint’s buffer and enable the endpoint. For interrupt OUT transfers, the device must enable the endpoint and retrieve received report data (usually signaled by a device interrupt) from the interrupt endpoint’s buffer.

**Retrieving descriptors**

The host computer identifies an attached USB peripheral when it retrieves a series of descriptors during enumeration. The enumeration process is triggered when a device is plugged into the bus or when the host powers up with a device attached.

Listing 1 shows a series of descriptors for a USB joystick. Every USB
A HID must also have one or more report descriptors, which are requested after the host has detected that the device is a HID and assigned the class driver to control it.

Windows’ Device Manager uses .inf files to decide which driver to assign to a device. HIDs can use the default file for HIDs (hiddev.inf for Windows 98, input.inf for Windows 2000), or you can provide an .inf file that contains your Vendor and Product IDs. The advantage of a custom .inf file is that you can include a device description and manufacturer name that will appear in the Control Panel, in place of the generic “Standard Device” listing.

Reports

A HID can send and receive reports using a device’s control endpoint and interrupt endpoint(s). HIDs don’t support USB’s bulk or isochronous transfer types.

Control transfers have no guaranteed delivery time. The host controller fits the transfers in as best it can. Ten percent of the bus bandwidth is reserved for control transfers, but the host may allocate some of the time to other devices.

Interrupt transfers have a guaranteed maximum latency, or time between transaction attempts. Each transaction carries a data packet. An interrupt endpoint can request a maximum latency from 1ms to 255ms, 10ms to 255ms for low-speed devices. For example, if the latency is 10ms, the host may initiate a new transaction anywhere from 1ms to 10ms after the last transaction attempt.

Which transfer type the host uses depends on the type of report requested and, sometimes, on the device hardware and version of Windows. A HID can exchange three types of reports: input, output, and feature. Input and output reports are best for data that must be sent or retrieved periodically, such as keypresses. Feature reports are best for data that transfers occasionally or isn’t time-critical—such as setup or configuration information.

When the host requests input reports, the device sends data to the host in interrupt transfers. The host
A HID can send and receive reports using a device’s control endpoint and interrupt endpoint(s).

schedules the transfer requests according to the maximum latency requested in the endpoint descriptor.

For an output report, the host sends data to a device using control or interrupt transfers. The ability to do interrupt OUT HID transfers was added in version 1.1 of the HID spec, and is available under Windows 98 SE and later Windows editions, including Windows 2000.

If the HID interface has an interrupt OUT endpoint, Windows 98 SE or later will use it for output reports, and Windows 98 Gold (original) will use control transfers. Device firmware would need to support both transfer types to enable a device to receive output reports either way. If the interface doesn’t have an interrupt OUT endpoint, the HID driver uses control transfers for output reports.

Feature reports can travel in either direction; they always use control transfers. To send an OUT feature report, the host sends a Set_Report request, followed by the report data, and the device returns status information to indicate the success or failure of the transfer. To receive an IN feature report, the host sends a Get_Report request, the device sends the report data, and the host returns status information to indicate the success or failure of the transfer.

If data is needed only occasionally and timing isn’t critical, feature reports keep the bus from clogging with periodic transfer attempts when data isn’t needed.

Another advantage of feature reports is the ability to support multiple report formats. The host can specify a report number in the control request. In contrast, in an interrupt transfer, the host just requests or sends data bytes without having the chance to name a specific report.

Report formats can be simple or complicated. Listing 2 shows a descriptor for a very basic report for a device that sends and receives two bytes.

Items in the report descriptor can specify how the data should be used (for example, which axis or button on a mouse), what units to apply, the range of valid values, and more. The items and values to use for common device types are defined in the HID specification and the HID Usage Tables document available from www.usb.org. A device that doesn’t fit a standard usage can use vendor-defined items.

Tools
To assist in developing HID firmware, the Implementers Forum provides two free tools: the Descriptor Tool and the USB Compliance Tool.

The Descriptor Tool (Figure 1) automates the process of building report descriptors and enables you to check the format before copying the descriptor into the device. You can build a descriptor by selecting items from a pick list and entering values (such as the number of bytes in the item) as needed. You can then ask the tool to check the descriptor and flag errors. To get you started, the tool comes with example descriptors for common device types.

The USB Compliance Tool is actually a suite of tools that performs a series of basic tests on any device, plus additional tests for HIDs, hubs, and communications-class devices.

The Compliance Tool loads its own diagnostic driver for the device under test. You can read and check descriptors and send other control requests. The HID tests (Figure 2) enable you to check the report descriptor and perform additional class-specific tests.

Windows communications
When you have the HID firmware up and running, you’re ready to start accessing the device from applications on the host computer. (This article concentrates on Windows, but HID support is also available or in progress for other operating systems.)
The Windows 98 and 2000 driver development kits (DDKs) include a fairly complete tutorial on user-mode (application-level) HID communications. The examples use C syntax, but with some translating you can use Visual BASIC, Delphi, or any other language that can call API functions. Table 1 lists the major API functions used in HID communications.

The first step in talking to any HID is finding it. This requires a series of API calls that steps through all attached HIDs, looking for one that matches your criteria. You can look for a standard device type like a joystick or keyboard, or you can look for a match with the Vendor and Product IDs or Product String in your descriptors.

The API functions that identify and return a device pathname for each HID are Windows Device Management Functions documented in the Windows NT and 2000 Platform Software Development Kits (SDKs), but not included in the Windows 98 DDK.

When you have a pathname, you can use CreateFile() to open a handle to the device, then use HID-specific API functions to read the Vendor and Product IDs, Product String, and other information about the device’s intended use and abilities.

Once you’ve identified the device you’re looking for, you’re ready to exchange reports using ReadFile() and WriteFile() (for input and output reports) or HidD_Get_Feature() and HidD_Set_Feature() (for feature reports).

One caution about input reports: ReadFile() is a blocking system call that won’t return until the device has returned the expected amount of report data. Using ReadFileEx() or overlapped I/O doesn’t help. So it’s best to place code that calls ReadFiles() in its own program thread, so it doesn’t hang the main application while waiting for data.

Between the support provided by Windows and other operating systems and the wide availability of USB ports, HIDs are worth a look. A HID is likely to be the quickest route to a finished PC peripheral, and you can be confident that you’ve chosen an interface that will be around for a long while.

Jan Axelson is the author of USB Complete: Everything You Need to Develop Custom USB Peripherals. Portions of this article are adapted from material in USB Complete. Jan hosts a USB Central page for developers at www.lvr.com/usb.htm.

### Resources


USBComp (www.usb.org/developers/tools.html). A free suite of tools for compliance testing.

DirectX Developers Center (msdn.microsoft.com/directx/). The DirectX DDK, newsgroups, articles, and more.

<table>
<thead>
<tr>
<th>API function</th>
<th>Documentation</th>
<th>Use in HID communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>HidD_GetHidGuid</td>
<td>Windows 98/2000 DDK</td>
<td>Returns the GUID associated with HIDs</td>
</tr>
<tr>
<td>SetupDiGetClassDevs</td>
<td>Windows NT/2000 DDK and NT/2000 Platform SDK</td>
<td>Returns an array of structures containing information about all installed HIDs</td>
</tr>
<tr>
<td>SetupDiEnumDeviceInterfaces</td>
<td>Windows NT/2000 DDK and NT/2000 Platform SDK</td>
<td>Returns a pointer to a structure that identifies an interface in the array returned by SetupDiGetClassDevs</td>
</tr>
<tr>
<td>SetupDiGetDeviceInterfaceDetail</td>
<td>Windows NT/2000 DDK and NT/2000 Platform SDK</td>
<td>Returns a device pathname for a specified device interface</td>
</tr>
<tr>
<td>CreateFile</td>
<td>Windows Platform SDK</td>
<td>Opens a handle to a HID using the pathname returned by SetupDiGetDeviceInterfaceDetail</td>
</tr>
<tr>
<td>HidD_GetAttributes</td>
<td>Windows 98/2000 DDK</td>
<td>Returns the Vendor ID, Product ID, and Version for a specified HID</td>
</tr>
<tr>
<td>HidD_GetPreparsedData</td>
<td>Windows 98/2000 DDK</td>
<td>Returns a handle to a buffer with information about a device's capabilities</td>
</tr>
<tr>
<td>HidD_GetCaps</td>
<td>Windows 98/2000 DDK</td>
<td>Returns a structure describing a device's capabilities</td>
</tr>
<tr>
<td>HidD_GetValueCaps</td>
<td>Windows 98/2000 DDK</td>
<td>Returns a structure describing the values in a device port</td>
</tr>
<tr>
<td>HidD_GetButtonCaps</td>
<td>Windows 98/2000 DDK</td>
<td>Returns a structure describing buttons in a device report</td>
</tr>
<tr>
<td>HidD_GetFeature</td>
<td>Windows 98/2000 DDK</td>
<td>Reads a feature report from a specified HID</td>
</tr>
<tr>
<td>HidD_SetFeature</td>
<td>Windows 98/2000 DDK</td>
<td>Sends a feature report to a specified HID</td>
</tr>
<tr>
<td>ReadFile</td>
<td>Windows Platform SDK</td>
<td>Reads in input report from a specified HID</td>
</tr>
<tr>
<td>WriteFile</td>
<td>Windows Platform SDK</td>
<td>Sends out output report to a specified HID</td>
</tr>
<tr>
<td>HidP_SetButtons</td>
<td>Windows 98/2000 DDK</td>
<td>Sets the button data in a report</td>
</tr>
</tbody>
</table>