11

Human Interface Devices: Using Control and Interrupt Transfers

The human interface device (HID) class was one of the first USB classes 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, many vendor-specific USB devices use the HID class.

Chapter 7 introduced the class. This chapter shows how to determine whether a specific device can fit into the human-interface class, details the firmware requirements that define a device as a HID and enable it to exchange data with its host, introduces the six HID-specific control requests, and presents an example of HID firmware. Chapter 12 describes

the reports that HIDs use to exchange information and Chapter 13 shows how to access HIDs from applications.

What is a HID?

The *human interface* in the name suggests that HIDs interact directly with people, and many HIDs do. A mouse may detect when someone presses a key or moves the mouse, or the host may send a message that translates to a joystick effect that the user experiences. Besides keyboards, mice, and joysticks, the HID class encompasses 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. The device just needs to be able to function within the limits of the HID class specification. These are the major abilities and limitations of HID-class devices:

- All data exchanged resides in structures called reports. 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, but each defined report has a fixed size.
- A HID interface must have an interrupt IN endpoint for sending Input reports.
- A HID interface can have at most one interrupt IN endpoint and one interrupt OUT endpoint. If you need more interrupt endpoints, you can create a composite device that contains multiple HIDs. An application must obtain a separate handle for each HID in the composite device.
- The interrupt IN endpoint enables the HID to send information to the host at unpredictable times. For example, there's no way for the computer to know when a user will press a key on the keyboard, so the host's driver uses interrupt transactions to poll the device periodically to obtain new data.
- The rate of data exchange is limited, especially at low and full speeds. As Chapter 3 explained, a host can guarantee a low-speed interrupt endpoint no more than 800 bytes/sec. For full-speed endpoints, the maxi-

mum is 64 kilobytes/sec., and for high-speed endpoints, the maximum is about 24 Megabytes/sec. if the host supports high-bandwidth endpoints and about 8 Megabytes/sec. if not. Control transfers have no guaranteed bandwidth except for the bandwidth reserved for all control transfers on the bus.

- Windows 98 Gold (original edition) supports USB 1.0, so interrupt OUT transfers aren't supported and all host-to-device reports must use control transfers.

Any device that can live within the class's limits is a candidate to be a HID. The HID specification mentions bar-code readers, thermometers, and voltmeters as examples of HIDs that might 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.

A HID interface may be just one of multiple USB interfaces supported by a device. For example, 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 on a device.

Hardware Requirements

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

Endpoints

All HID transfers use either the control endpoint or an interrupt endpoint. Every HID must have an interrupt IN endpoint for sending data to the host. An interrupt OUT endpoint is optional. Table 11-1 shows the transfer types and their typical uses in HIDs.

Table 11-1: 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	Typical 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

Reports

The requirement for an interrupt IN endpoint suggests that every HID must have at least one Input report defined in the HID's report descriptor. Output and Feature reports are optional.

Control Transfers

The HID specification defines six class-specific requests. Two requests, Set_Report and Get_Report, provide a way for the host and device to transfer reports to and from the device using control transfers. The host uses Set_Report to send reports and Get_Report to receive reports. The other four requests relate to configuring the device. The Set_Idle and Get_Idle requests set and read the Idle rate, which determines whether or not a device resends data that hasn't changed since the last poll. The Set_Protocol and Get_Protocol requests set and read a protocol value, which can enable a device to function with a simplified protocol when the full HID drivers aren't loaded on the host, such as during boot up.

Interrupt Transfers

Interrupt endpoints provide an alternate way of exchanging data, especially when the receiver must get the data quickly or periodically. Control transfers

can be delayed if the bus is very busy, while the bandwidth for interrupt transfers is guaranteed to be available when the device is configured.

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. Windows 98 SE was the first Windows edition to support USB 1.1 and HID 1.1.

Firmware Requirements

The device's firmware must also meet class requirements. The device's descriptors must include an interface descriptor that specifies the HID class, a HID descriptor, and an interrupt IN endpoint descriptor. An interrupt OUT endpoint descriptor is optional. The firmware must also contain a report descriptor that contains information about the contents of a HID's reports.

A HID can support one or more reports. The report descriptor specifies the size and contents of the data in a device's reports and may also include information about how the receiver of the data should use the data. Values in the descriptor define each report as an Input, Output, or Feature report. The host receives data in Input reports and sends data in Output reports. A Feature report can travel in either direction.

Every device should support at least one Input report that the host can retrieve using interrupt transfers or control requests. Output reports are optional. To be compatible with Windows 98 Gold, devices that use Output reports should support sending the reports using control transfers. Using interrupt transfers for Output reports is optional. Feature reports always use control transfers and are optional.

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 11-1 shows example device, configuration, interface, class, and endpoint descriptors for a vendor-specific HID. The host learns about the HID interface during enumeration by send-

```
{
// Device Descriptor

0x12,       // Descriptor size in bytes
0x01,       // Descriptor type (Device)
0x0200,     // USB Specification release number (BCD) (2.00)
0x00,       // Class Code
0x00,       // Subclass code
0x00,       // Protocol code
0x08,       // Endpoint 0 maximum packet size
0x0925,     // Vendor ID (Lakeview Research)
0x1234,     // Product ID
0x0100,     // Device release number (BCD)
0x01,       // Manufacturer string index
0x02,       // Product string index
0x00,       // Device serial number string index
0x01        // Number of configurations

// Configuration Descriptor

0x09,       // Descriptor size in bytes
0x02,       // Descriptor type (Configuration)
0x0029,     // Total length of this and subordinate descriptors
0x01,       // Number of interfaces in this configuration
0x01,       // Index of this configuration
0x00,       // Configuration string index
0xA0,       // Attributes (bus powered, remote wakeup supported)
0x50,       // Maximum power consumption (100 mA)

// Interface Descriptor

0x09,       // Descriptor size in bytes
0x04,       // Descriptor type (Interface)
0x00,       // Interface Number
0x00,       // Alternate Setting Number
0x02,       // Number of endpoints in this interface
0x03,       // Interface class (HID)
0x00,       // Interface subclass
0x00,       // Interface protocol
0x00,       // Interface string index
```

Listing 11-1: Descriptors for a vendor-specific HID (Sheet 1 of 2)

```
// HID Descriptor

0x09,       // Descriptor size in bytes
0x21,       // Descriptor type (HID)
0x0110,     // HID Spec. release number (BCD) (1.1)
0x00,       // Country code
0x01,       // Number of subordinate class descriptors
0x22,       // Descriptor type (report)
002F,       // Report descriptor size in bytes

// IN Interrupt Endpoint Descriptor

0x07,       // Descriptor size in bytes
0x05,       // Descriptor type (Endpoint)
0x81,       // Endpoint number and direction (1 IN)
0x03,       // Transfer type (interrupt)
0x40,       // Maximum packet size
0x0A,       // Polling interval (milliseconds)

// OUT Interrupt Endpoint Descriptor

0x07,       // Descriptor size in bytes
0x05,       // Descriptor type (Endpoint)
0x01,       // Endpoint number and direction (1 OUT)
0x03,       // Transfer type (interrupt)
0x40,       // Maximum packet size
0x0A        // Polling interval (milliseconds)
}
```

Listing 11-1: Descriptors for a vendor-specific HID (Sheet 2 of 2)

ing 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 requests the report descriptor and any physical descriptors.

The HID Interface

In the interface descriptor, bInterfaceclass = 3 to identify the interface as 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.

If bInterfaceSubclass = 1, the device supports a boot interface. A HID with a boot interface is usable when the host's HID drivers aren't loaded. This situation 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 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. If a device has a boot interface, the bInterfaceProtocol field indicates if the device supports the keyboard (1) or mouse (2) interface.

The HID Usage Tables document defines the report format for keyboards and mice that use the boot protocol. The BIOS knows what the boot protocol is and assumes that a boot device will support this protocol, so there's no need to read a report descriptor from the device. Before sending or requesting reports, the BIOS sends the HID-specific Set_Report request to request to use the boot protocol. When the full HID drivers have been loaded, the driver can use Set_Protocol to cause the device to switch from the boot protocol to the report protocol, which uses the report formats defined in the report descriptor

The bInterfaceSubclass field should equal zero if the HID doesn't support a boot protocol.

HID Class Descriptor

The HID class descriptor identifies additional descriptors for HID communications. The class descriptor has seven or more fields depending on the number of additional descriptors. Table 11-2 shows the fields. Note that the

Table 11-2: 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	This descriptor's type: 21h to indicate 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 report and physical descriptors.
6	bDescriptorType	1	The type of a class-specific descriptor that follows. (A report descriptor (required) is type 22h.)
7	wDescriptorLength	2	Total length of the descriptor identified above.
9	bDescriptorType	1	Optional. The type of a class-specific descriptor that follows. A physical descriptor is type 23h.
10	wDescriptorLength	2	Total length of the descriptor identified above. Present only if bDescriptorType is present immediately above. May be followed by additional wDescriptorType and wDescriptorLength fields to identify additional physical descriptors.

descriptor has two or more bDescriptorType fields. One identifies the HID descriptor and the other(s) identify the type of a subordinate descriptor.

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The value 21h indicates a HID descriptor.

The Class

bcdHID. The HID specification number that the interface complies with. In BCD format. 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 of a descriptor that is subordinate to the HID class descriptor. Every HID must contain a report descriptor. One or more physical descriptors are optional.

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

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

Report Descriptors

A report descriptor defines the format and use of the data in the HID's reports. If the device is a mouse, the data reports mouse movements and button clicks. If the device is a relay controller, the data specifies which relays to open and close.

A report descriptor needs to be flexible enough to handle devices with different purposes. The data should use a concise format to keep from wasting storage space in the device or bus time when the data transmits. HID report descriptors achieve both goals by using a format that's more complex and less readable than a more verbose format might be.

A report descriptor is a class-specific descriptor. The host retrieves the descriptor by sending a Get_Descriptor request with the wValue field containing 22h in the high byte.

Listing 11-2 is a bare-bones report descriptor that describes an Input report, an Output report, and a Feature report. The device sends two bytes of data in the Input report. The host sends two bytes of data in the Output report. The Feature report is two bytes that the host can send to the device or request from the device.

Each item in the report descriptor consists of a byte that identifies the item and one or more bytes containing the item's data. The HID class specifica-

```
0x06  0xFFA0          Usage Page (vendor-defined)
0x09  0x01            Usage (vendor-defined)
0xA1  0x01            Collection (Application)

0x09  0x03            Usage (vendor-defined)
0x15  0x00            Logical Minimum (0)
0x26  0x00FF          Logical Maximum (255)
0x95  0x02            Report Count (2)
0x75  0x08            Report Size (8 bits)
0x81  0x02            Input (Data, Variable, Absolute)

0x09  0x04            Usage (vendor-defined)
0x15  0x00            Logical Minimum (0)
0x26  0x00FF          Logical Maximum (255)
0x75  0x08            Report Count (2)
0x95  0x02            Report Size (8 bits)
0x91  0x02            Output (Data, Variable, Absolute)

0x09  0x05            Usage (vendor-defined)
0x15  0x00            Logical Minimum (0)
0x26  0x00FF          Logical Maximum (255)
0x75  0x08            Report Count (2)
0x95  0x02            Report Size (8 bits)
0xB1  0x02            Feature (Data, Variable, Absolute)

0xC0                  End Collection
```

Listing 11-2: This report descriptor defines an Input report, an Output report, and a Feature report. Each report transfers two vendor-defined bytes.

tion defines items that a report can contain. 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. In the example descriptor, the Usage Page is the vendor-defined value FFA0h. 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 an individual report in a 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 01h.

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.

The Collection contains three reports. Each report has these items:

A vendor-defined Usage applies to the data in the report.

A Logical Minimum and Logical Maximum specify the range of values that the report can contain.

The Report Count item indicates how many data items the report contains. In the example, each report contains two data items.

The Report Size item indicates how many bits are in each reported data item. In the example, each data item is eight bits.

The final item specifies whether the report is an Input report (81h), Output report (91h), or Feature report (B1h). The bits provided with the item contain additional information about the report data.

The End Collection item closes the Application Collection.

Chapter 12 has more about report formats.

HID-specific Requests

The HID specification defines six HID-specific requests. Table 11-3 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, and Set_Idle) are optional except that a keyboard using the boot protocol must support Set_Idle. If a HID doesn't have an Interrupt OUT endpoint or if the HID is communicating with a 1.0 host such as Windows 98 Gold, a HID that wants to receive reports from the host must support Set_Report.

Table 11-3: The HID class defines six HID-specific requests.

Request Number	Request	Data Source (Data stage)	wValue (high byte, low byte)	wIndex	Data Length (bytes) (wLength)	Data Stage Contents	Required?
01h	Get_ Report	device	report type, report ID	interface	report length	report	yes
02h	Get_ Idle	device	0, report ID	interface	1	idle duration	no
03h	Get_ Protocol	device	0	interface	1	protocol	required for HIDs that support a boot protocol
09h	Set_ Report	host	report type, report ID	interface	report length	report	no
0Ah	Set_ Idle	no Data stage	idle duration, report ID	interface	–	–	no, except for keyboards using the boot protocol
0Bh	Set_ Protocol	no Data stage	0, protocol	interface	–	–	required for HIDs that support a boot protocol

Get_Report

Purpose: The host requests an Input or Feature report from a HID using a control transfer.

Request Number (bRequest): 01h

Source of Data: device

Data Length (wLength): length of the report

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

Contents of wIndex field: the number of the interface the request is directed to.

Contents of data packet in the Data stage: the report

Comments: All HIDs must support this request. See also Set_Report

Get_Idle

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

Request Number (bRequest): 02h

Source of Data: device

Data Length (wLength): 1

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

Contents of wIndex 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.

Get_Protocol

Purpose: The host learns whether the boot or report protocol is currently active in the HID.

Request Number (bRequest): 03h

Source of Data: device

Data Length (wLength): 1

Contents of wValue field: 0

Contents of wIndex 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. See also Set_Protocol.

Set_Report

Purpose: The host sends an Output or Feature report to a HID using a control transfer.

Request Number (bRequest): 09h

Source of Data: host

Data Length (wLength): length of the report

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

Contents of wIndex field: the number of the interface the request is directed to.

Contents of data packet in the Data stage: the report

Comments: If a HID interface 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 HID. HIDs aren't required to support this request. See also Get_Report.

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 (bRequest): 0Ah

Source of Data: no Data stage

Data Length (wLength): no Data stage

Contents of wValue field: The high byte sets the duration, or the maximum amount of time between reports. A value of zero means that the HID will send a report only when the report data has changed. The low byte indicates the report ID that the request applies to. If the low byte is zero, the request applies to all of the HID's Input reports.

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

Contents of data packet in the Data stage: no Data stage.

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 Input report sent, on receiving an interrupt IN token packet, the HID sends a report. If the data hasn't changed and the duration time hasn't elapsed since the last report, the HID returns NAK. If the data hasn't changed and the duration time has elapsed since the last report, the HID sends a report. A duration value of zero indicates an infinite duration: the HID sends a report only if the report data has changed and responds to all other interrupt IN requests with NAK.

If the HID returns a STALL in response to this request, the HID can send reports whether or not the data has changed. On enumerating a HID, the Windows HID driver attempts to set the idle rate to zero. The HID should Stall this request if an infinite Idle duration isn't wanted! HIDs aren't required to support this request except for keyboards using the boot protocol.

See also Get_Idle.

Set_Protocol

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

Request Number (bRequest): 0Bh

Source of Data: no Data stage

Data Length (wLength): no Data stage

Contents of wValue field: the protocol (0 = boot protocol, 1 = report protocol).

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

Contents of data packet in the Data stage: no Data stage

Comments: Boot devices must support this request. See also Get_Protocol

Transferring Data

When enumeration is complete, the host has identified the device interface as a HID and has established pipes with the interface's endpoints and learned what report formats to use to send and receive data.

The host can then request reports using either interrupt IN transfers and/or control transfers with Get_Report requests. The device also has the option to support receiving reports using interrupt OUT transfers and/or control transfers with Set_Report requests.

If you don't have example firmware for Get_Report and Set_Report, enumeration code can serve as a model. Get_Report is a control Read transfer that behaves in a similar way to Get_Descriptor except that the device returns a report instead of a descriptor. Set_Report is a control Write transfer. Unfortunately, the only standard USB request with a host-to-device Data stage is the rarely supported Set_Descriptor, so example code for control Write transfers is harder to find.

About the Example Code

The example code in this chapter is written for the Microchip PIC18F4550 introduced in Chapter 6. The code is based on Microchip's USB Firmware Framework. The Framework code is for the PIC18 family of microcontrollers but can give an idea of how to structure code for other CPUs. My code is adapted from Microchip's mouse code and implements a generic HID device that exchanges reports in both directions.

Portions of the code can be useful even if your device isn't a HID. The control-transfer examples can serve as models for responding to other class-specific and vendor-specific requests. From the firmware's point of view, bulk and interrupt transfers are identical, so the interrupt-transfer code can serve as a model for any firmware that uses bulk or interrupt transfers.

In the Framework code, a group of system files handles general USB tasks and class-specific tasks. These files typically require no changes or only minor changes and additions for specific applications. The system files include these:

usbmmap.c allocates memory for variables, endpoints, and other buffers used in USB communications.

usbdrv.c contains functions to detect device attachment and removal, check and respond to USB hardware interrupts, enter and exit the Suspend state, and respond to bus resets.

usbctrltrf.c contains functions for handling transactions in control transfers. The functions decode received Setup packets, manage the sending and receiving of data in the Data stage, and manage the sending and receiving of status information in the Status stage.

usb9.c contains functions that manage responding to the requests defined in Chapter 9 of the USB specification. The functions decode received requests, provide pointers to descriptors and other requested data to return, and take requested actions such as selecting a configuration or setting an address.

hid.c contains functions that manage tasks that are specific to the HID class. The functions decode received Setup data directed to the HID interface, define pointers to data to be sent and locations to store received data in control and interrupt transfers, and respond to other HID-specific requests such as Get_Idle and Set_Idle.

Additional files handle other tasks:

usbdesc.c contains structures that hold the device's descriptors, including the device descriptor, configuration and subordinate descriptors, string descriptors, and report descriptor. This information will of course differ for every device.

user_generic_hid.c contains most of the code that is specific to the generic HID application. The functions obtain the data that the device will send in reports and use the data received in reports.

main.c initializes the system and executes a loop that checks the bus status and calls functions in the *user_generic_hid* file to carry out the device's purpose.

The sections that follow contain excerpts from this code. The excerpts concentrate on the application-specific code used to send and receive reports.

The complete device firmware and host applications to communicate with the device are available from *www.Lvr.com*.

See Chapter 6 for an introduction to the '18F4550's architecture.

Sending Reports via Interrupt Transfers

When the HID driver is loaded on the host, the host controller begins sending periodic IN token packets to the HID's interrupt IN endpoint. The endpoint should NAK these packets until the HID has an Input report to send. To send data in an interrupt transfer, device firmware typically places the data in the endpoint's buffer and configures the endpoint to send the data to the host on receiving an IN token packet.

The code below executes after the host has configured the HID. The *usbmmap.c* file declares hid_report_in as a char array whose length equals the endpoint's wMaxPacketSize:

```
extern volatile far unsigned char
  hid_report_in[HID_INT_IN_EP_SIZE];
```

The code initializes the interrupt IN endpoint by setting values in the endpoint's buffer descriptor (HID_BD_IN):

```
// Set the endpoint's address register to the address
// of the hid_report_in buffer.

HID_BD_IN.ADR = (byte*)&hid_report_in;

// Set the status register bits:
// _UCPU = the CPU owns the buffer.
// _DAT1 = a DATA1 data toggle is expected next.
// (Before sending data, call the mUSBBufferReady
// macro to toggle the data toggle.)

HID_BD_IN.Stat._byte = _UCPU|_DAT1;
```

Listing 11-3 is a function that accepts a pointer to a buffer containing report data (*buffer) and the number of bytes to send (len) and makes the data available in the endpoint's buffer.

```
void HIDTxReport(char *buffer, byte len)
{
    byte i;

    // len can be no larger than the endpoint's wMaxPacketSize
    // (HID_INT_IN_EP_SIZE). Trim len if necessary

    if(len > HID_INT_IN_EP_SIZE)
        len = HID_INT_IN_EP_SIZE;

    // Copy data from the passed buffer to the endpoint's buffer.

    for (i = 0; i < len; i++)
        hid_report_in[i] = buffer[i];

    // To send len bytes, set the buffer descriptor's count
    // register to len.

    HID_BD_IN.Cnt = len;

    // Toggle the data toggle and transfer ownership of the
    // buffer to the SIE, which will send the data on receiving
    // an interrupt IN token packet.

    mUSBBufferReady(HID_BD_IN);

}//end HIDTxReport
```

Listing 11-3: The HIDTxReport function provides data to send in an Input report at the HID's interrupt IN endpoint and prepares the endpoint to send the report.

This code calls the function:

```
char transmit_buffer[2];

// Place report data in transmit_buffer:

transmit_buffer[0] = 104;
transmit_buffer[1] = 105;

// If necessary, wait until the CPU owns the interrupt
// IN endpoint's buffer.

while(mHIDTxIsBusy())
{
    // Service USB interrupts.

    USBDriverService();
}

// Make the report data available to send in the next
// interrupt IN transaction.

HIDTxReport(transmit_buffer, 2);
```

Before calling the function, the firmware places the report data in a char array (transmit_buffer) and waits if necessary for the mHIDTxIsBusy macro to return false, indicating that the CPU owns the HID's interrupt IN endpoint buffer. While waiting, the loop calls the USBDriverService function to service any USB interrupts that occur.

In the HIDTxReport function, if the len value passed to the function is greater than the endpoint's wMaxPacketSize value, len is trimmed to wMaxPacketSize. The function copies the report data from the passed buffer to the endpoint's buffer (hid_report_in).

The endpoint's byte-count register (HID_BD_IN.Cnt) holds the number of bytes to send. The macro mUSBBufferReady toggles the data-toggle bit and transfers ownership of the endpoint buffer to the SIE. On receiving an interrupt IN token packet, the SIE sends the packet and returns ownership of the buffer to the CPU.

Receiving Reports via Interrupt Transfers

If the HID interface has an interrupt OUT endpoint and the operating system supports USB 1.1 or later, the host can send Output reports to the device using interrupt transfers.

To receive data in an interrupt transfer, device firmware typically configures the endpoint to receive data from the host on receiving an OUT token packet, and an interrupt or polled register announces that data has been received.

The code below executes after the host has configured the HID. The *usbmmap.c* file declares hid_report_out as a char array whose length equals the endpoint's wMaxPacketSize:

```
extern volatile far unsigned char
  hid_report_out[HID_INT_OUT_EP_SIZE];
```

The code initializes the interrupt OUT endpoint by setting values in the endpoint's buffer descriptor (HID_BD_OUT):

```
// Set the endpoint's byte-count register to the
// length of the output report expected.

HID_BD_OUT.Cnt = sizeof(hid_report_out);

// Set the endpoint's address register to the address
// of the hid_report_out buffer.

HID_BD_OUT.ADR = (byte*)&hid_report_out;

// Set the status register bits:

// _USIE =  The SIE owns the buffer.
// _DATA0 = a DATA0 data toggle is expected next.
// _DTSEN = Enable data-toggle synchronization.

HID_BD_OUT.Stat._byte = _USIE|_DAT0|_DTSEN;
```

The endpoint is then ready to receive a report. Listing 11-4 is a function that retrieves data that has arrived at an interrupt OUT endpoint. The function accepts a pointer to a buffer to copy the report data to (*buffer) and the

```
byte HIDRxReport(char *buffer, byte len)
{
    // hid_rpt_rx_len is a byte variable declared in hid.c
    hid_rpt_rx_len = 0;

    if(!mHIDRxIsBusy())
    {
        // If necessary, trim len to equal
        // the actual number of bytes received.

        if(len > HID_BD_OUT.Cnt)
            len = HID_BD_OUT.Cnt;

        // The report data is in hid_report_out.
        // Copy the data to the user's buffer (buffer).

        for(hid_rpt_rx_len = 0;
            hid_rpt_rx_len < len;
            hid_rpt_rx_len++)

            buffer[hid_rpt_rx_len] =
                hid_report_out[hid_rpt_rx_len];

        // Prepare the endpoint buffer for next OUT transaction.

        // Set the endpoint's count register to the length of
        // the report buffer.

        HID_BD_OUT.Cnt = sizeof(hid_report_out);

        // The mUSBBufferReady macro toggles the data toggle
        // and transfers ownership of the buffer to the SIE.

        mUSBBufferReady(HID_BD_OUT);

    }//end if

        return hid_rpt_rx_len;

}//end HIDRxReport
```

Listing 11-4: The HIDRxReport function retrieves data received in an Output report at the HID's interrupt OUT endpoint.

number of bytes expected (len). The function returns the number of bytes copied or zero if no data is available.

This code calls the function:

```
byte number_of_bytes_read;
char receive_buffer[2];

number_of_bytes_read =
    HIDRxReport(receive_buffer, 2);
```

In the HIDRxReport function, the mHIDRxIsBusy macro checks to see if the CPU has ownership of the buffer. If not, the function returns zero. Otherwise, if necessary, the function trims the expected number of bytes (len) to match the number of bytes received as reported in the endpoint's byte-count register (HID_BD_OUT.Cnt). The received report is in the buffer hid_report_out. The function copies the received bytes from hid_report_out to the buffer that was passed to the function (receive_buffer). To prepare for a new transaction, the function sets the endpoint's byte-count register to the size of the hid_report_out buffer and calls the mUSBBufferReady macro, which toggles the data toggle and transfers ownership of the buffer to the SIE. The endpoint can then receive new data. When the function returns, the report data is available in receive_buffer. The hid_rpt_rx_len variable contains the number of bytes received (zero if no bytes were received).

Sending Reports via Control Transfers

To send an Input or Feature report using a control transfer, device firmware must detect the Get_Report code in the bRequest field of the Setup stage of a control request directed to the HID interface. The firmware then configures Endpoint 0 to send the report in the Data stage and receive the host's response in the Status stage.

Microchip's Framework HID example includes code that detects the request and calls the HIDGetReportHandler function in Listing 11-5.

The function examines the high byte of the wValue field of the Setup stage's data packet (MSB(SetupPkt.W_Value)) to determine if the host is request-

```
void HIDGetReportHandler(void)
{
    // The report type is in the high byte of the setup packet's
    // wValue field. 1 = Input; 3 = Feature.

    switch(MSB(SetupPkt.W_Value))
    {
        byte count;

        case 1: // Input report

            // Find out which report ID was specified.
            // The report ID is in the low byte (LSB) of the
            // wValue field. This example supports Report ID 0.
            switch(LSB(SetupPkt.W_Value))
            {
                case 0: // Report ID 0

                // The HID class code will handle the request.
                ctrl_trf_session_owner = MUID_HID;

                // Provide the data to send.
                GetInputReport0();

                break;

                case 1: // Report ID 1
                  // Add code to handle Report ID 1 here.
                break;

            } // end switch(LSB(SetupPkt.W_Value))

            break;

        case 3: // Feature report

            // Add code to handle Feature reports here.

    } // end switch(MSB(SetupPkt.W_Value))
}//end HIDGetReportHandler
```

Listing 11-5: The HIDGetReportHandler function is called on receiving a Get_Report request.

ing an Input or Feature report. The low byte of the wValue field (LSB(SetupPkt.W_Value)) is the report ID that names the specific report requested.

If the code supports the requested report, the ctrl_trf_session_owner variable is set to MUID_HID. The firmware handling other stages of the transfer can use this variable to detect who is handling the request.

To send Input report 0, the function calls the GetInputReport0 function (Listing 11-6). This function sets pSrc.bRam, a pointer to data in RAM, to the location of the report's data (hid_report_in).

The Framework firmware prepares Endpoint 0 to send the data that pSrc.bRam points to on receiving an IN token packet and accepts the host's zero-length packet in the Status stage of the transfer.

The code for sending other reports, including Input reports with other report IDs and Feature reports, is much the same except that the source of the report's contents will change depending on the report.

Receiving Reports via Control Transfers

To receive an Output or Feature report in a control transfer, the firmware must detect the Set_Report request in the bRequest field of a request directed to the HID interface. The device must also receive the report data in the Data stage and return a handshake in the Status stage. Microchip's Framework HID example includes code that detects the request and calls the HIDSetReportHandler function in Listing 11-7.

The function examines the high byte of the wValue field of the Setup stage's data packet (MSB(SetupPkt.W_Value)) to determine if the host is sending an Output or Feature report. The low byte of the wValue field (LSB(SetupPkt.W_Value)) is the report ID that names the specific report being sent.

If the code supports the requested report, the ctrl_trf_session_owner variable is set to MUID_HID so the firmware that handles other stages of the transfer can detect who is handling the request.

The Framework firmware prepares Endpoint 0 to receive the report data on receiving an OUT token packet.

```
void GetInputReport0(void)
{
    byte count;

    // Set pSrc.bRam to point to the report.

    pSrc.bRam = (byte*)&hid_report_in;
}
```

Listing 11-6: The GetInputReport0 function handles report-specific tasks.

When Output report 0 arrives in the Data stage of the transfer, the Framework firmware makes the data available at the RAM location pointed to by pDst.bRam and manages the sending of a zero-length packet in the transfer's Status stage. In this example, pDst.bRam has been set to point to the char array hid_report_out. The firmware can call a function that uses the report data:

```
if (ctrl_trf_session_owner == MUID_HID)
{
  // Call a function that uses the report data.

  HandleControlOutReport();
}
```

The code for receiving other reports, including Output reports with other report IDs and Feature reports, is much the same except that the firmware's use of the report's contents will change depending on the report.

```
void HIDSetReportHandler(void)
{
    // The report type is in the high byte of the Setup packet's
    // wValue field. 2 = Output; 3 = Feature.

    switch(MSB(SetupPkt.W_Value))
    {
        case 2: // Output report

            // The report ID is in the low byte of the Setup
            // packet's wValue field.
            // This example supports Report ID 0.

            switch(LSB(SetupPkt.W_Value))
            {
                case 0: // Report ID 0

                  // The HID-class code will handle the request.
                    ctrl_trf_session_owner = MUID_HID;

                   // When the report arrives in the Data stage,
                    // the report data will be available in
                    // hid_report_out.
                    pDst.bRam = (byte*)&hid_report_out;
                    break;

                case 1: // Report ID 1
               // Place code to handle Report ID 1 here.

            } // end switch(LSB(SetupPkt.W_Value))

            break;

        case 3: // Feature report

            // Place code to handle Feature reports here.

    } // end switch(MSB(SetupPkt.W_Value))

} //end HIDSetReportHandler
```

Listing 11-7: The HIDSetReportHandler function prepares to receive an Output or Feature report from the host.