USB Human Interface Devices (HID) are devices that, like the name suggests, allow an interface that lets humans interact with the computer. Common examples include USB mice, USB keyboards, USB joysticks, and other such devices. The protocol used by USB HID devices is defined in the USB HID specification. Some chipsets support emulating USB keyboards and mice as standard PS/2 devices, but many chipsets don't. Thus, a USB HID driver is necessary in some PCs that may not have PS/2 ports at all.

Protocol

USB HID devices are mainly based on two protocols: the report protocol and the boot protocol. A report is a data structure that is sent to the host from the device, or can also be sent from the host to the device. When a device sends a report to the host, it normally contains status change information, such as a keypress, mouse movement, etc. When the host sends a report to the device, it normally contains commands for configuring the device, such as setting LEDs on a keyboard, for example. This protocol of course depends on the standard USB framework. USB HID devices communicate using interrupt transfers, as they don't always transfer data, but when they do, they require very fast response from the software as well as the data transferred is normally small. Reports generally have two types, depending on the protocol type. Report protocols are based on the concept of "items," and their structures are defined in report descriptors. The boot protocol is much simpler, and follows standard structures for mice and keyboards. For simplicity's sake, this article will discuss the boot protocol only, for now at least.

Detecting HID devices

HID devices have class/sub-class values of both zeroes in their device descriptors, and instead have the class/sub-class values valid in their interface descriptors. Keep in mind that interface descriptors cannot be manually requested, and must be acquired along with the configuration and endpoint descriptors. The class value that identifies a HID device in the interface descriptors is 3. The sub-class value in the interface descriptor can be either 1 to indicate the device supports the boot protocol, or zero to indicate the device only supports the report protocol. The protocol field in the interface descriptor as well determines it is a mouse or a keyboard. Specifically, 1 indicates the HID device is a keyboard, while 2 indicates the HID device is a mouse.

"SetProtocol" request

Assuming a USB HID device supports the boot protocol, as explained in the section above, where is has a class value of 3 and a sub-class value of 1, the driver software can select the protocol to use. It uses the "SetProtocol" request to tell the device whether it wants to use the report protocol or the boot protocol. For simplicity's sake, this article will describe the boot protocol only, for now. To send the "SetProtocol" request, software sends a regular SETUP transaction to the device's control endpoint zero. The SETUP packet's request type would contain 0x21, the request code for "SetProtocol" is 0x0B, and the value field of the SETUP packet should contain 0 to indicate boot protocol, or 1 to indicate report protocol. The index and length fields both must be zero, as the index is unused and this request has no data stages. This command is only supported on device that support the boot protocol at all. After this command has been used, all reports sent from the device to the host will be either boot reports or regular reports, depending on the type the software requests.

"GetReport" request

The software can request a report from a USB device using the control endpoint and the regular SETUP packet. The SETUP packet's request type would contain 0xA1, the request code for "GetReport" is 1, the SETUP packet's "value" field would contain 0x0100, to request an input packet with ID zero, and the length field would be the length of the data stage the host wishes to receive. The SETUP packet should be sent to the device endpoint 0 (control endpoint.) For keyboards, the data stage is normally 8 bytes, while for mice, the data stage has the first 3 bytes in a standard format, while the rest of it may be used by device-specific

features. Receiving reports this way is recommended only to test that the device initialization was completed successfully, or such, and the "GetReport" request should not be used to poll the HID device for changes, as the SETUP and STATUS stages waste too much time. Instead, software should poll the HID using interrupt transfers, using the interrupt IN endpoint.

Interrupt endpoint

It is generally recommended that the HID reports to the software using interrupt transfers, and that software should normally avoid the "GetReport" request mentioned above. Driver software should request the configuration descriptor for the HID device. A HID device must support at least one configuration. Software should then scan the endpoint descriptors, searching for a descriptor that indicates an "interrupt IN" type, which is an endpoint that sends device data to the host using interrupt transfers. Software should save the 4-bit ID of the endpoint, as well as the 8-bit interval of the endpoint. The interval value encodes time in milliseconds (ms) in which timespace the software should poll for a report packet once. For example, if the interval value is 8, software should request a report from the device every 8 ms. If software requests a report too early, for example, after 6 ms, the device may send the same packet as before, or it may not send anything, and return NAK instead. If software requests a report after the timespan, say for example 9 ms, then the device will send the new packet. Software constantly polls USB HID devices using this described method. This is also a good chance to optimize your USB code, as the polling functions will be run hundreds of times per second. Specifically in Bochs, the interval value of the USB mouse is 10 ms, and so software polls the USB device 100 times per second.

USB keyboard

USB keyboards communicate with software using reports, just like other HID devices. USB keyboards are detected by having a class code of 3 and a protocol value of 1, in the interface descriptor. I will be describing the boot protocol here, for simplicity's sake, for now at least.

Report format

This report must be requested by the software using interrupt transfers once every interval milliseconds, and the interval is defined in the interrupt IN descriptor of the USB keyboard. The USB keyboard report may be up to 8 bytes in size, although not all these bytes are used and it's OK to implement a proper implementation using only the first three or four bytes (and this is how I do it.) Just for completion's sake, however, I will describe the full report mechanism of the keyboard. Notice that the report structure defined below applies to the boot protocol only.

Offset	Size	Description
0	Byte	Modifier keys status.
1	Byte	Reserved field.
2	Byte	Keypress #1.
3	Byte	Keypress #2.
4	Byte	Keypress #3.
5	Byte	Keypress #4.
6	Byte	Keypress #5.
7	Byte	Keypress #6.

Modifier keys status: This byte is a bitfield, where each bit corresponds to a specific modifier key. When a bit is set to 1, the corresponding modifier key is being pressed. Unlike PS/2 keyboards, USB keyboards don't have "scancodes" for modifier keys. The bit structure of this byte is:

Bit	Bit Length	Description
0	1	Left Ctrl.
1	1	Left Shift.
2	1	Left Alt.

3	1	Left GUI (Windows/Super key.)
4	1	Right Ctrl.
5	1	Right Shift.
6	1	Right Alt.
7	1	Right GUI (Windows/Super key.)

When software receives an interrupt and, for example, one of the Shift modifier keys are set to 1, software should use the scancode table for the shift modification to get the key from the scancode.

Reserved field: This byte is reserved by the USB HID specification, and thus software should ignore it.

Keypress fields: One keyboard report can indicate up to 6 keypresses. All these values are unsigned 8-bit values (unlike PS/2 scancodes, which are mostly 7-bit) which indicate the key being pressed. A reference on the USB scancode to ASCII character conversion table is in the bottom of the article.

Keypress mechanism

USB keyboards send interrupts when a key is pressed or released, just like a PS/2 keyboard. However, unlike PS/2 keyboards, USB keyboard does not have the concept of "make" and "break" scancodes. When a user presses a key, the interrupt comes in with a scancode value in one of the keypress fields. When a key is released, the corresponding keypress field is returned zero in the next packet. To illustrate this more clearly and to illustrate why there are more than one keypress scancode field, let's look at the following examples. Assume the user pressed the "A" key, which is scancode 0x04. The returned interrupt packet would look like this:

```
00 00 04 00 00 00 00 00
```

Notice the modifier keys are zero, because the user isn't pressing any. The reserved field is also zero, as recommended by the USB HID spec. The first keypress field contains 0x04, which corresponds to the "A" key. Now, let's assume the user lets go of the "A" key. The packet sent would look like:

```
00 00 00 00 00 00 00 00
```

Now, let's assume the user pressed the "A" key, and then pressed the "B" key (scancode 0x05) without letting go of the "A" key:

```
00 00 04 05 00 00 00 00
```

Notice how one interrupt packet is capable of transferring two keypresses together. Now let's assume the user presses the "C" key (scancode 0x06) without letting go of either "A" or "B" key:

```
00 00 04 05 06 00 00 00
```

Now what if the user lets go of the "A" key but keeps pressing the "B" and "C" keys? The keyboard will indicate that "A" is no longer being pressed, and "B" and "C" will move towards the beginning of the packet:

```
00 00 05 06 00 00 00 00
```

As is probably obvious now, the USB keyboard returns scancodes in order of which one was pressed first. As such, if the first keypress field is zero, there are no keys that send scancodes being pressed. If it is not zero, software can check the next fields, to see if another key is being pressed as well.

The concept of modifier keys is probably obvious, but just for completion, let's assume the user pressed the left shift key with the "X" key (scancode 0x1B). The interrupt packet sent will contain:

```
02 00 1B 00 00 00 00 00
```

Notice that bit 1 (value 0x02) of the modifier field is set, to indicate that the the left shift key is being pressed.

There is also a "phantom condition" which you can think of as an overflow. A USB keyboard packet can indicate up to 6 keypresses in one transfer, but let's imagine someone dropped their keyboard and more than

6 keys were pressed in one time. The keyboard will enter the phantom condition, in which all reported keys will be of the invalid scancode 0x01. Modifier keys will still be reported however. Imagine that 8 keys (or any random number more than 6) are being pressed, and the right shift key is also being pressed. The packet sent would look like:

```
20 00 01 01 01 01 01 01
```

Notice that the modifier keys are still being indicated, but the actual scancodes all return the phantom condition. There are other special scancodes besides the phantom condition: 0x00 indicates there is no scancode and no key being pressed, 0x01 indicates the phantom condition we just explained, 0x02 indicates the keyboard's self-test failed and 0x03 indicates an undefined error occured. Starting from 0x04 the scancodes are valid and correspond to "real" keys. IMHO, a device driver should ignore the phantom condition if it happens.

Auto-repeat

One thing that makes USB keyboards a pain is that there is no mechanism for auto-repeat and auto-repeat delays in hardware; this must be implemented entirely in software, unlike PS/2 keyboards. For an example auto-repeat delay of 500 milliseconds and an auto-repeat speed of 10 characters per second, when software becomes aware that a certain key is being pressed without being released constantly, it ignores all the key presses except the first one for 500 milliseconds (or whatever auto-repeat delay you like), and if this times out and the key is still being pressed, the driver will report 10 keypresses every 1 second, or whatever auto-repeat speed you want as well. This would make the user feel natural, just like you would when you press and hold a key; the first key appears on the screen, the computer waits a short delay, and then the keys keep coming up.

LED lamps

LED lamps are also handled in software, and according to the hardware, NumLock, CapsLock and ScrollLock are normal keys that send normal scancodes. The driver is responsible for manipulating the LED lamps when one of these keys are pressed.

To set the LED lamps, the driver sends a SetReport request to the device using a standard USB Setup Transaction, with a one-byte data stage. The setup packet's request type should contain 0x21, the request code for SetReport is 0x09. The value field of the setup packet contains the report ID in the low byte, which should be zero. The high byte contains the report type, which should be 0x02 to indicate an output report, or a report that is being sent from the software to the hardware. The index field should contain the interface number of the USB keyboard, which is the number present in the interface descriptor which indicated this device was a USB keyboard at all. The data stage should be 1 byte, which is a bitfield. This Setup Transaction should be transferred to control endpoint zero, which would work on all hardware. Other hardware may or may not support the optional interrupt OUT endpoint. If the hardware supports the interrupt OUT endpoint, you can just transfer the 1 byte data stage to the interrupt OUT endpoint, without the extra overhead of the SETUP stage and STATUS stage. If the hardware support the interrupt OUT endpoint, you should avoid the control endpoint when possible, as the interrupt OUT endpoint is faster and can be programmed with interrupt transfers instead of setup transfers. The format of the 1-byte data stage (for SETUP transaction) or 1-byte interrupt OUT transfer is shown below. When a bit is set to 1, the corresponding LED is turned on.

Bit	Bit Length	Description
0	1	Num Lock.
1	1	Caps Lock.
2	1	Scroll Lock.
3	1	Compose.
4	1	Kana.
5	3	Reserved, must be zero.

USB mouse

USB mice, just like any other HID device, communicate with the software using reports, which are sent via interrupt endpoints or can be manually requested with the "GetReport" request. USB mice have a protocol value of 2 in the interface descriptor.

Report format

This report must be requested by the host using interrupt transfers once every interval milliseconds. Interval is defined in the interrupt IN descriptor of the USB mouse device. Only the first three bytes of the USB mouse report are defined. The remaining bytes, if existed, may be used for device-specific features. Software may request only three bytes in an interrupt transfer, and this will not cause an error even if the actual packet is larger. The table below defines the report format for USB mice operating using the boot protocol.

Offset	Size	Description
0	Byte	Button status.
1	Byte	X movement.
2	Byte	Y movement.

Button status: This byte is a bitfield, in which the lowest three bits are standard format. The remaining 5 bits may be used for device-specific purposes.

Bit	Bit Length	Description
0	1	When set to 1, indicates the left mouse button is being clicked.
1	1	When set to 1, indicates the right mouse button is being clicked.
2	1	When set to 1, indicates the middle mouse button is being clicked.
3	5	These bits are reserved for device-specific features.

X movement: This is a signed 8-bit integer that represents the X movement. Bit 7 (value 0x80) determines the sign of the value. When this value is negative, the mouse is being moved to the left. When this value is positive, the mouse is being moved to the right. Notice that unlike PS/2 mice, the movement values for USB mice are 8-bit signed integers and not 9-bit integers.

Y movement: This is also a signed 8-bit integer that represents the Y movement. When this value is negative, the mouse is being moved up. When the value is positive, the mouse is being moved down (towards the user.)

USB Report Protocol

This is the complicated one supported by all HID class devices. Mouses, keyboards, joysticks...

The interface descriptor will contain an HID descriptor alongside the endpoint descriptors:

Offset	Field	Size	Type	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	HID Descriptor Type (0x22)
2	bcdHID	2	BCD	HID Class Specification Release Number in Binary-Coded Decimal (i.e, 1.10 is expressed as 110h).
4	bCountryCode	1	ID	Country code of localised hardware (0 if irrelevant. Mainly for keyboards).
5	bNumDescriptors	1	Number	Number of descriptors >= 1
3i+6	bDescriptorType	1	Number	Type of HID descriptor
3i+7	wDescriptorLength	2	Number	Length of HID Descriptor

There is always one descriptor, a Report Descriptor, followed by n Optional Descriptors. The types and lengths only are in an array at the end of the overarching HID Descriptor.

HID Descriptor Types

Value	Description
21h	HID Descriptor
22h	Report Descriptor
23h	Physical Descriptor

To get the details of these descriptors, use the standard USB GET_DESCRIPTOR, but with an interface recipient.

bmRequestType	bRequest	wValue		wIndex	wLength
10000001b	GET_DESCRIPTOR 6	Descriptor Type	Descriptor Index	Interface Number	Descriptor Length

Descriptor Index is 0 except for Physical Descriptors, where Index 0 will enumerate descriptor sets and their sizes.

Country Codes

Value	Country	Value	Country	Value	Country	Value	Country	Value	Country
00h	*Not Supported*	08h	French	10h	Korean	18h	Slovakia	20h	UK
01h	Arabic	09h	German	11h	Latin American	19h	Spanish	21h	US
02h	Belgian	0Ah	Greek	12h	Netherlands	1Ah	Swedish	22h	Yugoslavia
03h	Canadian-Bilingual	0Bh	Hebrew	13h	Norwegian	1Bh	Swiss/ French	23h	Turkish-F
04h	Canadian-French	0Ch	Hungary	14h	Persian	1Ch	Swiss/ German		
05h	Czechia	0Dh	International (ISO)	15h	Poland	1Dh	Switzerland		
06h	Danish	0Eh	Italian	16h	Portuguese	1Eh	Taiwan		
07h	Finnish	0Fh	Japan (Katakana)	17h	Russia	1Fh	Turkish-Q		

Report Descriptor

This is not a value table, length and content vary as required.

It's a sequential list of items. Items come in two basic types, short and long.

Short Item

Offset	Field	Size	Type	Description
0:0	bSize	2 bits	Enum	Size of optional data in bytes
0:2	bType	2 bits	Enum	Type of this descriptor
0:4	bTag	4 bits	Number	Function of the item
1	Data	bSize	Data	Item data

Note that as a special case a bSize of 3 corresponds to 4 bytes.

bType:

Value	Description
00h	Main
01h	Global

02h	Local
03h	Reserved

Long Item

Offset	Field	Size	Type	Description
0:0	bSize	2 bits	Enum	Size = 2
0:2	bType	2 bits	Enum	Type = 3 (Reserved)
0:4	bTag	4 bits	Number	1111b - Long
1	bDataSize	1	Number	Byte count of data
2	bLongItemTag	1	Number	Long item tag
3	Data	bDataSize	Number	Data

Parsing

The report descriptor's items are parsed in a sequential manner. The parser is a state machine. A complete report descriptor may look something like this:

```
static const uint8_t hidReportDescriptor [] =
{
    0x05, 0x01,    // UsagePage(Generic Desktop[1])
    0x09, 0x04,    // UsageId(Joystick[4])
    0xA1, 0x01,    // Collection(Application)
    0x85, 0x01,    //     ReportId(1)
    0x09, 0x01,    //     UsageId(Pointer[1])
    0xA1, 0x00,    //     Collection(Physical)
    0x09, 0x30,    //         UsageId(X[48])
    0x09, 0x31,    //         UsageId(Y[49])
    0x15, 0x80,    //         LogicalMinimum(-128)
    0x25, 0x7F,    //         LogicalMaximum(127)
    0x95, 0x02,    //         ReportCount(2)
    0x75, 0x08,    //         ReportSize(8)
    0x81, 0x02,    //         Input(Data, Variable, Absolute, NoWrap, Linear,
PreferredState, NoNullPosition, BitField)
    0x05, 0x09,    //         UsagePage(Button[9])
    0x19, 0x01,    //         UsageIdMin(Button 1[1])
    0x29, 0x03,    //         UsageIdMax(Button 3[3])
    0x15, 0x00,    //         LogicalMinimum(0)
    0x25, 0x01,    //         LogicalMaximum(1)
    0x95, 0x03,    //         ReportCount(3)
    0x75, 0x01,    //         ReportSize(1)
    0x81, 0x02,    //         Input(Data, Variable, Absolute, NoWrap, Linear,
PreferredState, NoNullPosition, BitField)
    0xC0,          //     EndCollection()
    0x05, 0x02,    //     UsagePage(Simulation Controls[2])
    0x09, 0xBB,    //     UsageId(Throttle[187])
    0x15, 0x80,    //     LogicalMinimum(-128)
    0x25, 0x7F,    //     LogicalMaximum(127)
    0x95, 0x01,    //     ReportCount(1)
    0x75, 0x08,    //     ReportSize(8)
    0x81, 0x02,    //     Input(Data, Variable, Absolute, NoWrap, Linear,
PreferredState, NoNullPosition, BitField)
    0x75, 0x05,    //     ReportSize(5)
    0x81, 0x03,    //     Input(Constant, Variable, Absolute, NoWrap, Linear,
PreferredState, NoNullPosition, BitField)
    0xC0,          // EndCollection()
};
```

Recall the descriptor types.

Main - This is added to the logical tree. There are 5 subtypes:

- *Input* - 1000 00 nn
- *Output* - 1001 00 nn
- *Feature* - 1011 00 nn
- *Collection* - 1010 00 nn
- *End Collection* - 1100 00 nn

Global - Adjusts the global state machine. New main items will inherit this state. This is useful where multiple axes are similar, for instance.

- *Usage Page* - 0000 01 nn
- *Logical Minimum* - 0001 01 nn
- *Logical Maximum* - 0010 01 nn
- *Physical Minimum* - 0011 01 nn
- *Physical Maximum* - 0100 01 nn
- *Unit Exponent* - 0101 01 nn
- *Unit* - 0110 01 nn
- *Report Size* - 0111 01 nn
- *Report ID* - 1000 01 nn
- *Report Count* - 1001 01 nn
- *Push* - 1010 01 nn
- *Pop* - 1011 01 nn

Local - Do not carry over to next main item.

- *Usage* - 0000 10 nn
- *Usage Minimum* - 0001 10 nn
- *Usage Maximum* - 0010 10 nn
- *Designator Index* - 0011 10 nn
- *Designator Minimum* - 0100 10 nn
- *Designator Maximum* - 0101 10 nn
- *String Index* - 0111 10 nn
- *String Minimum* - 1000 10 nn
- *String Maximum* - 1001 10 nn
- *Delimiter* - 1010 10 nn

Note that locals can apply to more than one control is a single item.

Physical Descriptors

To quote the HID specification:
"Note Physical Descriptors are entirely optional. They add complexity and offer very little in return for most devices. However, some devices, particularly those with a large number of identical controls (for example, buttons) will find that Physical Descriptors help different applications assign functionality to these controls in a more consistent manner. Skip the following section if you do not plan on supporting Physical Descriptors."