4

Enumeration: How the Host Learns about Devices

Before applications can communicate with a device, the host needs to learn about the device and assign a device driver. Enumeration is the exchange of information that accomplishes these tasks. The process includes assigning an address to the device, reading descriptors from the device, assigning and loading a device driver, and selecting a configuration that specifies the device's power requirements, endpoints, and other features. The device is then ready to transfer data using any of the endpoints in its configuration.

This chapter describes the enumeration process, including the structure of the descriptors that the host reads from the device during enumeration. You don't need to know every detail about enumeration in order to design a USB peripheral, but understanding how enumeration works in general is essential

in creating the descriptors that will reside in the device and in writing firmware that responds to enumeration requests.

The Process

One of the duties of a hub is to detect the attachment and removal of devices. Each hub has an interrupt IN endpoint for reporting these events to the host. On system boot-up, the host polls its root hub to learn if any devices are attached, including additional hubs and devices attached to those hubs. After boot-up, the host continues to poll periodically to learn of any newly attached or removed devices.

On learning of a new device, the host sends a series of requests to the device's hub, causing the hub to establish a communications path between the host and the device. The host then attempts to enumerate the device by sending control transfers containing standard USB requests to the device's Endpoint 0. All USB devices must support control transfers, the standard requests, and Endpoint 0. For a successful enumeration, the device must respond to each request by returning requested information and taking other requested actions.

From the user's perspective, enumeration is invisible and automatic except for possibly a message that announces the detection of a new device and whether the attempt to configure it succeeded. Sometimes on first use, the user needs to assist in selecting a driver or specifying where the host should look for driver files.

When enumeration is complete, Windows adds the new device to the Device Manager's display in the Control Panel. When a user removes a device from the bus, Windows removes the device from the Device Manager.

In a typical device, firmware contains the information the host will request, and a combination of hardware and firmware decodes and responds to requests for the information. Some controllers can manage the enumeration entirely in hardware, with no firmware support. On the host side, under

Windows there's no need to write code for enumerating because the operating system handles the process.

Enumeration Steps

The USB specification defines six device states. During enumeration, a device moves through four of the states: Powered, Default, Address, and Configured. (The other states are Attached and Suspend.) In each state, the device has defined capabilities and behavior.

The steps below are a *typical* sequence of events that occurs during enumeration under Windows. But device firmware *must not assume* that the enumeration requests and events will occur in a particular order. To function successfully, a device must detect and respond to any control request or other bus event at any time.

1. The user attaches a device to a USB port. Or the system powers up with a device already attached. The port may be on the root hub at the host or a hub that connects downstream from the host. The hub provides power to the port, and the device is in the Powered state.

2. The hub detects the device. The hub monitors the voltages on the signal lines of each of its ports. The hub has a pull-down resistor of 14.25 to 24.8 kilohms on each of the port's two signal lines (D+ and D-). A device has a pull-up resistor of 900 to 1575 ohms on either D+ for a full-speed device or D- for a low-speed device. High-speed-capable devices attach at full speed. When a device plugs into a port, the device's pull-up brings its line high, enabling the hub to detect that a device is attached. Chapter 15 has more on how hubs detect devices.

On detecting a device, the hub continues to provide power but doesn't yet transmit USB traffic to the device.

3. The host learns of the new device. Each hub uses its interrupt endpoint to report events at the hub. The report indicates only whether the hub or a port (and if so, which port) has experienced an event. On learning of an event, the host sends the hub a Get_Port_Status request to find out more. Get_Port_Status and the other requests described here are standard

hub-class requests that all hubs support. The information returned tells the host when a device is newly attached.

4. The hub detects whether a device is low or full speed. Just before the hub resets the device, the hub determines whether the device is low or full speed by examining the voltages on the two signal lines. The hub detects the speed of a device by determining which line has the higher voltage when idle. The hub sends the information to the host in response to the next Get_Port_Status request. A 1.x hub may instead detect the device's speed just after a bus reset. USB 2.0 requires speed detection to occur before the reset so the hub knows whether to check for a high-speed-capable device during reset, as described below.

5. The hub resets the device. When a host learns of a new device, the host controller sends the hub a Set_Port_Feature request that asks the hub to reset the port. The hub places the device's USB data lines in the Reset condition for at least 10 milliseconds. Reset is a special condition where both D+ and D- are a logic low. (Normally, the lines have opposite logic states.) The hub sends the reset only to the new device. Other hubs and devices on the bus don't see the reset.

6. The host learns if a full-speed device supports high speed. Detecting whether a device supports high speed uses two special signal states. In the Chirp J state, only the D+ line is driven and in the Chirp K state, only the D- line is driven.

During the reset, a device that supports high speed sends a Chirp K. A high-speed-capable hub detects the chirp and responds with a series of alternating Chirp Ks and Chirp Js. On detecting the pattern KJKJKJ, the device removes its full-speed pull up and performs all further communications at high speed. If the hub doesn't respond to the device's Chirp K, the device knows it must continue to communicate at full speed. All high-speed devices must be capable of responding to enumeration requests at full speed.

7. The hub establishes a signal path between the device and the bus. The host verifies that the device has exited the reset state by sending a Get_Port_Status request. A bit in the returned data indicates whether the

device is still in the reset state. If necessary, the host repeats the request until the device has exited the reset state.

When the hub removes the reset, the device is in the Default state. The device's USB registers are in their reset states and the device is ready to respond to control transfers at Endpoint 0. The device communicates with the host using the default address of 00h. The device can draw up to 100 milliamperes from the bus.

8. The host sends a Get_Descriptor request to learn the maximum packet size of the default pipe. The host sends the request to device address 0, Endpoint 0. Because the host enumerates only one device at a time, only one device will respond to communications addressed to device address 0, even if several devices attach at once.

The eighth byte of the device descriptor contains the maximum packet size supported by Endpoint 0. A Windows host requests 64 bytes, but after receiving just one packet (whether or not it has 64 bytes), the host begins the Status stage of the transfer. On completion of the Status stage, a Windows host requests the hub to reset the device, as in Step 5 above. The USB specification doesn't require a reset here. Resetting is a precaution that ensures that the device will be in a known state when the reset ends.

9. The host assigns an address. The host controller assigns a unique address to the device by sending a Set_Address request. The device completes the Status stage of the request using the default address and then implements the new address. The device is now in the Address state. All communications from this point on use the new address. The address is valid until the device is detached, the port is reset, or the system reboots. On the next enumeration, the host may assign a different address to the device.

10. The host learns about the device's abilities. The host sends a Get_Descriptor request to the new address to read the device descriptor. This time the host retrieves the entire descriptor. The descriptor is a data structure containing the maximum packet size for Endpoint 0, the number of configurations the device supports, and other basic information about the device. The host uses this information in the communications that follow.

The host continues to learn about the device by requesting the one or more configuration descriptors specified in the device descriptor. A request for a configuration descriptor is actually a request for the configuration descriptor followed by all of that descriptor's subordinate descriptors. A Windows host begins by requesting just the configuration descriptor's nine bytes. Included in these bytes is the total length of the configuration descriptor and its subordinate descriptors.

Windows then requests the configuration descriptor again, this time using the retrieved total length. The device responds by sending the configuration descriptor followed by the configuration's interface descriptor(s), with each interface descriptor followed by any endpoint descriptors for the interface. Some configurations also include class- or vendor-specific descriptors that extend or modify another descriptor. These descriptors follow the descriptor being extended or modified. Each descriptor begins with its length and type. The Descriptors section in this chapter has more on what each descriptor contains.

11. The host assigns and loads a device driver (except for composite devices). After learning about a device from its descriptors, the host looks for the best match in a device driver to manage communications with the device. In selecting a driver, Windows tries to match the information in the PC's INF files with the Vendor ID, Product ID, and (optional) release number retrieved from the device. If there is no match, Windows looks for a match with any class, subclass, and protocol values retrieved from the device. If the device has been enumerated previously, Windows can use information in the system registry instead of searching the INF files. After the operating system assigns and loads the driver, the driver may request the device to resend descriptors or send other class-specific descriptors.

An exception to this sequence is composite devices, which can have different drivers assigned to different interfaces in a configuration. The host can assign these drivers only after the interfaces are enabled, which requires the device to be configured (as described in the next step).

12. The host's device driver selects a configuration. After learning about a device from the descriptors, the device driver requests a configuration by

sending a Set_Configuration request with the desired configuration number. Some devices support only one configuration. If a device supports multiple configurations, the driver can decide which configuration to request based on information the driver has about how the device will be used, or the driver can ask the user what to do or just select the first configuration. The device reads the request and enables the requested configuration. The device is now in the Configured state and the device's interface(s) are enabled.

For composite devices, the host assigns drivers at this point. As with other devices, the host uses the information retrieved from the device to find a matching driver for each active interface in the configuration. The device is now ready for use.

The other two device states are Attached and Suspend.

Attached state. If the hub isn't providing power to a device's VBUS line, the device is in the Attached state. The absence of power may occur if the hub has detected an over-current condition or if the host requests the hub to remove power from the port. With no power on VBUS, the host and device can't communicate, so from their perspective, the situation is the same as when the device isn't attached at all.

Suspend State. A device enters the Suspend state after detecting no bus activity, including Start-of-Frame markers, for at least 3 milliseconds. In the Suspend state, the device should limit its use of bus power. Both configured and unconfigured devices must support this state. Chapter 16 has more about the Suspend state.

Enumerating a Hub

Hubs are also USB devices, and the host enumerates a newly attached hub in exactly the same way as other devices. If the hub has devices attached, the host enumerates each of these after the hub informs the host of their presence.

Device Removal

When a user removes a device from the bus, the hub disables the device's port. The host learns that the removal occurred after polling the hub, learning that an event has occurred, and sending a Get_Port_Status request to find out what the event was. Windows removes the device from the Device Manager's display and the device's address becomes available to another newly attached device.

Tips for Successful Enumeration

Successful enumeration is essential. Without it, the device and host can't perform any additional communications. Most chip vendors provide example code to get you started. Even if your device uses a different class or has other differences, the example code can serve as a model. If your controller interfaces to an external CPU, you may have to adapt code written for another chip.

In general, a device should assume nothing about what requests or events the host will initiate and should just concentrate on responding to requests and events as they occur. The following tips have specific advice about how to avoid common problems.

Don't assume requests or events will occur in a specific order. The USB 2.0 specification says nothing about what order a host might choose in sending control requests during enumeration. A host might also choose to reset the bus at any time, and the device must detect the reset and respond appropriately.

Be ready to abandon a control transfer or end it early. On receiving a new Setup packet, a device must abandon any transfer in progress and begin the new one. On receiving an OUT token packet, the device must assume that the host is beginning the Status stage of the transfer even if the device hasn't sent all of the requested data in the Data stage.

Don't attempt to send more data than the host asks for. In the Data stage of a Control Read transfer, a device should send no more than the amount

of data the host has asked for. If the host requests nine bytes, the device should send no more than nine bytes.

Send a zero-length data packet when required. If the device has less than the requested amount of data to return and if the amount of data is an exact multiple of the endpoint's maximum packet size, the device should indicate that there is no more data by returning a zero-length data packet in response to the next IN token packet.

Stall unsupported requests. A device shouldn't assume it knows every request the host might send. The device should return a STALL in response to any request the device doesn't recognize or support.

Don't set the address too soon. In a Set_Address request, the device should set its new address only after the Status stage of the request is complete.

Be ready to enter the Suspend state. A host can suspend the bus when the device is in any powered state, including before the device has been configured. When the bus is suspended, the device must reduce its use of bus power.

Test under different host-controller types. Some host controllers schedule multiple stages of a control transfer in a single frame, while others don't. Devices should be able to handle either way. Chapter 8 has more about host controllers.

Descriptors

USB descriptors are the data structures, or formatted blocks of information, that enable the host to learn about a device. Each descriptor contains information about the device as a whole or an element of the device.

All USB devices must respond to requests for the standard USB descriptors. The device must store the information in the descriptors and respond to requests for the descriptors.

Types of Descriptors

As described earlier in this chapter, during enumeration the host uses control transfers to request descriptors from the device. As enumeration progresses, the requested descriptors concern increasingly small elements of the device: first the entire device, then each configuration, each configuration's interface(s), and finally each interface's endpoint(s). Table 4-1 lists the descriptor types.

The higher-level descriptors inform the host of any additional, lower-level descriptors. Except for compound devices, each device has one and only one device descriptor that contains information about the device as a whole and specifies the number of configurations the device supports. Each device also has one or more configuration descriptors that contain information about the device's use of power and the number of interfaces supported by the configuration. Each interface descriptor specifies zero or more endpoint descriptors that contain the information needed to communicate with an endpoint. Each endpoint descriptor has information about how the endpoint transfers data. An interface with no endpoint descriptors must use the control endpoint for communications.

On receiving a request for a configuration descriptor, a device should return the configuration descriptor and all of the configuration's interface, endpoint, and other subordinate descriptors, up to the requested number of bytes. There is no request to retrieve, for example, only an endpoint descriptor. Devices that support both full and high speeds support two additional descriptor types: device_qualifier and other_speed_configuration. These and their subordinate descriptors contain information about the device's behavior when using the speed not currently selected.

A string descriptor can store text such as the vendor's or device's name. Other descriptors can contain index values that point to these string descriptors, and the host can read the string descriptors using Get_Descriptor requests.

In addition to the standard descriptors, a device may contain class- or vendor-specific descriptors. These descriptors offer a structured way for a device to provide more detailed information about itself. For example, an interface

Table 4-1: The bDescriptorType field in a descriptor contains a value that identifies the descriptor type.

bDescriptorType	Descriptor Type	Required?
01h	device	Yes.
02h	configuration	Yes.
03h	string	No. Optional descriptive text.
04h	interface	Yes.
05h	endpoint	No, if the device uses only Endpoint 0.
06h	device_qualifier	Yes, for devices that support both full and high speeds. Not allowed for other devices.
07h	other_speed_configuration	Yes, for devices that support both full and high speeds. Not allowed for other devices.
08h	interface_power	No. Supports interface-level power management.
09h	OTG	For On-The-Go devices only.
0Ah	debug	No.
0Bh	interface_association	For composite devices.

descriptor may specify that the interface belongs to the HID class and has a HID class descriptor.

Each descriptor contains a value that identifies the descriptor type. Table 4-1 shows the values for the standard descriptor types. In addition to these values, a class or vendor may define additional descriptors. Two examples of class codes are 29h for a hub descriptor and 21h for a HID descriptor. Within the HID class, 22h indicates a report descriptor and 23h indicates a physical descriptor.

In the descriptor's bDescriptorType value, bit 7 is always zero. Bits 6 and 5 identify the descriptor type: 00h=standard, 01h=class, 02h=vendor, 03h=reserved. Bits 4 through 0 identify the descriptor.

Each descriptor consists of a series of fields. Most of the field names use prefixes to indicate something about the format or contents of the data in that field: *b* = byte (8 bits), *w* = word (16 bits), *bm* = bit map, *bcd* = binary-coded decimal, *i* = index, *id* = identifier.

Device Descriptor

The device descriptor contains basic information about the device. This descriptor is the first one the host reads on device attachment and includes the information the host needs to retrieve additional information from the device. A host retrieves a device descriptor by sending a Get_Descriptor request with the high byte of the Setup transaction's wValue field equal to 1.

The descriptor has 14 fields. Table 4-2 lists the fields in the order they occur in the descriptor. The descriptor includes information about the descriptor itself, the device, its configurations, and any classes the device belongs to. The following descriptions group the information by function.

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The constant DEVICE (01h).

The Device

bcdUSB. The USB specification version that the device and its descriptors comply with in BCD (binary-coded decimal) format. If you think of the version's value as a decimal number, the upper byte represents the integer, the next four bits are tenths, and the final four bits are hundredths. So version 1.0 is 0100h; version 1.1 is 0110h, and version 2.0 is 0200h. Note that version 1.1 is *not* 0101h. Also remember that a 2.0 device does not have to be high speed. Any new low- or full-speed design should comply with the latest version of the specification.

idVendor. Members of the USB-IF and others who pay an administrative fee receive the rights to use a unique Vendor ID. The host may have an INF file that contains this value, and if so, Windows uses the value to help decide what driver to load for the device. Except for devices used only in-house where the user is responsible for preventing conflicts, every device descriptor must have a valid Vendor ID in this field.

idProduct. The owner of the Vendor ID assigns a Product ID to identify the device. Both the device descriptor and the device's INF file on the host may contain this value, and if so, Windows uses the value to help decide

Table 4-2: The device descriptor has 14 fields in 18 bytes.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant DEVICE (01h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize0	1	Maximum packet size for Endpoint 0
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number (BCD)
14	iManufacturer	1	Index of string descriptor for the manufacturer
15	iProduct	1	Index of string descriptor for the product
16	iSerialNumber	1	Index of string descriptor containing the serial number
17	bNumConfigurations	1	Number of possible configurations

what driver to load for the device. Each Product ID is specific to a Vendor ID, so multiple vendors can use the same Product ID without conflict.

bcdDevice. The device's release number in BCD format. The vendor assigns this value. The host may use this value in deciding which driver to load.

iManufacturer. An index that points to a string describing the manufacturer. This value is zero if there is no manufacturer descriptor.

iProduct. An index that points to a string describing the product. This value is zero if there is no string descriptor.

iSerialNumber. An index that points to a string containing the device's serial number. This value is zero if there is no serial number. Some device classes (such as mass storage) require serial numbers. Serial numbers are useful if users may have more than one identical device on the bus and the host needs to keep track of which is which even after rebooting. Serial numbers

also enable a host to determine whether a peripheral is the same one used previously or a new installation of a peripheral with the same Vendor ID and Product ID. No devices with the same Vendor ID, Product ID, and device release number should have the same serial number.

The Configuration

bNumConfigurations. The number of configurations the device supports.

bMaxPacketSize0. The maximum packet size for Endpoint 0. The host uses this information in the requests that follow. For low-speed devices, this value must be 8. Full-speed devices may use 8, 16, 32, or 64. High-speed devices must use 64.

Classes

bDeviceClass. For devices whose function is defined at the device level, this field specifies the device's class. Values from 1 to FEh are reserved for USB's defined classes. Table 4-3 shows the defined codes. The value FFh means that the class is specific to the vendor and defined by the vendor. Many devices specify their class or classes in interface descriptors, and for these devices, the bDeviceClass field in the device descriptor is 00h (or EFh if the function uses an interface association descriptor).

bDeviceSubclass. This field can specify a subclass within a class. If bDeviceClass is 0, the bDeviceSubclass must be 0. If bDeviceClass is between 1 and FEh, bDeviceSubclass must be a code defined in a USB class specification. A value of FFh means that the subclass is specific to the vendor. A subclass can add support for additional features and abilities shared by a group of functions within a class.

bDeviceProtocol. This field can specify a protocol defined by the selected class or subclass. For example, a 2.0 hub uses this field to indicate whether the hub is currently supporting high speed and if so, if the hub supports one or multiple transaction translators. If bDeviceClass is between 01h and FEh, the protocol must be a code defined by a USB class specification.

Table 4-3: The bDeviceClass field in the device descriptor can name a class the device belongs to.

bDeviceClass	Description
00h	The interface descriptor names the class. (Use EFh if the function has an interface association descriptor.)
02h	Communications
09h	Hub
DCh	Diagnostic device (can also be declared at interface level) bDeviceSubClass = 1 for Reprogrammable Diagnostic Device with bDeviceProtocol = 1 for USB2 Compliance Device
E0h	Wireless Controller (can also be declared at interface level) bDeviceSubClass = 1 for RF Controller with bDeviceProtocol = 1 for Bluetooth Programming Interface
EFh	Miscellaneous Device bDeviceSubClass = 2 for Common Class with bDeviceProtocol = 1 for Interface Association Descriptor
FFh	Vendor-specific (can also be declared at interface level)

Device_qualifier Descriptor

Devices that support both full and high speeds must have a device_qualifier descriptor. When a device switches speeds, some fields in the device descriptor may change. The device_qualifier descriptor contains the values of these fields at the speed not currently in use. In other words, the contents of fields in the device and device_qualifier descriptors swap depending on which speed is being used. A host retrieves a device_qualifier descriptor by sending a Get_Descriptor request with the high byte of the Setup transaction's wValue field equal to 6.

The descriptor has nine fields. Table 4-4 lists the fields in the order they occur in the descriptor. The descriptor includes information about the descriptor itself, the device, its configurations, and its classes.

The Vendor ID, Product ID, device release number, manufacturer string, product string, and serial-number string don't change when the speed changes, so the device_qualifier descriptor doesn't include these values.

The host can use a Get_Descriptor request to retrieve the device_qualifier descriptor. The following descriptions group the information by function.

Table 4-4: The device_qualifier descriptor has nine fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant DEVICE_QUALIFIER (06h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize0	1	Maximum packet size for Endpoint 0
8	bNumConfigurations	1	Number of possible configurations
9	Reserved	1	For future use

The Descriptor

bLength. The length in bytes of the descriptor.

bDescriptorType. The constant DEVICE_QUALIFIER (06h).

The Device

bcdUSB. The USB specification number that the device and its descriptors comply with. Must be at least 0200h (USB 2.0).

The Configuration

bNumConfigurations. The number of configurations the device supports.

bMaxPacketSize0. The maximum packet size for Endpoint 0.

Classes

bDeviceClass. For devices that belong to a class, this field can name the class.

bDeviceSubclass. For devices that belong to a class, this field can specify a subclass within the class.

bDeviceProtocol. This field can specify a protocol defined by the selected class or subclass. For example, a 2.0 hub must support both a low- and full-speed protocol and a high-speed protocol. The device descriptor con-

tains the code for the currently active protocol, and the device_qualifier descriptor contains the code for the not-active protocol.

Reserved. For future use.

Configuration Descriptor

After retrieving the device descriptor, the host can retrieve the device's configuration, interface, and endpoint descriptors.

Each device has at least one configuration that specifies the device's features and abilities. Often a single configuration is enough, but a device with multiple uses or modes can support multiple configurations. Only one configuration is active at a time. Each configuration requires a descriptor. The configuration descriptor contains information about the device's use of power and the number of interfaces supported. Each configuration descriptor has subordinate descriptors, including one or more interface descriptors and optional endpoint descriptors. A host retrieves a configuration descriptor and its subordinate descriptors by sending a Get_Descriptor request with the high byte of the Setup transaction's wValue field equal to 2.

The host selects a configuration with the Set_Configuration request and reads the current configuration number with a Get_Configuration request.

The descriptor has eight fields. Table 4-5 lists the fields in the order they occur in the descriptor. The fields contain information about the descriptor itself, the configuration, and the device's use of power in that configuration. The following descriptions group the information by function.

The Descriptor

bLength. The length (in bytes) of the descriptor.

bDescriptorType. The constant CONFIGURATION (02h).

wTotalLength. The number of bytes in the configuration descriptor and all of its subordinate descriptors.

Table 4-5: The configuration descriptor has eight fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Configuration (02h)
2	wTotalLength	2	The number of bytes in the configuration descriptor and all of its subordinate descriptors
4	bNumInterfaces	1	Number of interfaces in the configuration
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests
6	iConfiguration	1	Index of string descriptor for the configuration
7	bmAttributes	1	Self/bus power and remote wakeup settings
8	bMaxPower	1	Bus power required, expressed as (maximum milliamperes/2)

The Configuration

bConfigurationValue. Identifies the configuration for Get_Configuration and Set_Configuration requests. Must be 1 or higher. A Set_Configuration request with a value of zero causes the device to enter the Not Configured state.

iConfiguration. Index to a string that describes the configuration. This value is zero if there is no string descriptor.

bNumInterfaces. The number of interfaces in the configuration. The minimum is 1.

Power Use

bmAttributes. Bit 6=1 if the device is self-powered or 0 if bus-powered. Bit 5=1 if the device supports the remote wakeup feature, which enables a suspended USB device to tell its host that the device wants to communicate. A USB device must enter the Suspend state if there has been no bus activity for 3 milliseconds. If an event at a suspended device requires action from the host, a device with remote wakeup enabled can request the host to resume communications.

The other bits in the field are unused. Bits 0 through 4 must be 0. Bit 7 must be 1. (In USB 1.0, bit 7 was set to 1 to indicate that the configuration was bus powered. In USB 1.1 and higher, setting bit 6 to 0 is enough to indicate that the configuration is bus powered.)

bMaxPower. Specifies how much bus current a device requires. The bMaxPower value equals *one half* the number of milliamperes required. If the device requires 200 milliamperes, bMaxPower=100. The maximum current a device can request is 500 milliamperes. Storing half the number of milliamperes enables one byte to store values up to the maximum. If the requested current isn't available, the host will refuse to configure the device. A driver may then request an alternate configuration if available.

Other_speed_configuration Descriptor

The other descriptor unique to devices that support both full and high speeds is the other_speed_configuration descriptor. The structure of the descriptor is identical to that of the configuration descriptor. The only difference is that the other-speed_configuration_descriptor describes the configuration when the device is operating at the speed not currently active. The other_speed_configuration descriptor has subordinate descriptors just as the configuration descriptor does. A host retrieves an other_speed_configuration descriptor by sending a Get_Descriptor request with the high byte of the Setup transaction's wValue field = 7.

The descriptor has eight fields. Table 4-6 lists the fields in the order they occur in the descriptor.

Interface Association Descriptor

An interface association descriptor (IAD) identifies multiple interfaces that are associated with a function. In relation to a device and its descriptors, the term interface refers to a feature or function a device implements.

Most device classes specify their functions at the interface level rather than at the device level. Assigning functions to interfaces makes it possible for a single configuration to support multiple interfaces and thus multiple functions. As explained in Chapter 1, a device that has multiple interfaces that are

Table 4-6: The other_speed_configuration descriptor has the same eight fields as the configuration descriptor.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant OTHER_SPEED_CONFIGURATION (07h)
2	wTotalLength	2	The number of bytes in the configuration descriptor and all of its subordinate descriptors
4	bNumInterfaces	1	Number of interfaces in the configuration
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests
6	iConfiguration	1	Index of string descriptor for the configuration
7	bmAttributes	1	Self/bus power and remote wakeup settings
8	MaxPower	1	Bus power required, expressed as (maximum milliamperes/2)

active at the same time is a composite device. Each interface has its own interface descriptor and an endpoint descriptor for each endpoint the interface uses. The host loads a driver for each interface.

When two or more interfaces in a configuration are associated with the same function, the interface association descriptor can tell the host which interfaces are associated with each other. For example, a video-camera function may use one interface to control the camera and another to carry the video data.

The USB Engineering Change Notice that defines the interface association descriptor says that the descriptor "must be supported by future implementations of devices that use multiple interfaces to manage a single device function." Devices that comply with the video-class specification must use an interface association descriptor. Class specifications that predate the descriptor of course don't require it. Hosts that don't support the descriptor ignore it. Support for the descriptor was added in Windows XP SP2.

To enable the host to identify devices that use the Interface Association descriptor, the device descriptor should contain the following values: bDeviceClass = EFh (miscellaneous device class), bDeviceSubClass = 02h (com-

mon class), and bDeviceProtocol = 01h (interface association descriptor). These codes are together referred to as the "Multi-interface Function Device Class Codes."

A host retrieves an interface association descriptor by requesting the configuration descriptor for the configuration the interface association belongs to.

An interface association descriptor has eight fields. Table 4-8 lists the fields in the order they occur in the descriptor. The following descriptions group the information by function.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant INTERFACE ASSOCIATION (0Bh).

The Interfaces

bFirstInterface. Identifies the interface number of the first interface of multiple interfaces associated with a function. The interface number is the value of bInterfaceNumber in the interface descriptor. The interface numbers of associated interfaces must be contiguous.

bInterfaceCount. Gives the number of contiguous interfaces associated with the function.

The Function

bFunctionClass. A class code for the function shared by the associated interfaces. For classes that don't specify a value to use, the preferred value is the bInterfaceClass value from the descriptor of the first associated interface. Values from 01h to FEh are reserved for USB-defined classes. FFh indicates a vendor-defined class. Zero is not allowed.

bFunctionSubClass. A subclass code for the function shared by the associated interfaces. For classes that don't specify a value to use, the preferred value for existing device classes is the bInterfaceSubClass value from the descriptor of the first associated interface.

Table 4-7: The interface association descriptor has eight fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Interface Association (0Bh)
2	bFirstInterface	1	Number identifying the first interface associated with the function
3	bInterfaceCount	1	The number of contiguous interfaces associated with the function
4	bFunctionClass	1	Class code
5	bFunctionSubClass	1	Subclass code
6	bFunctionProtocol	1	Protocol code
8	iFunction	1	Index of string descriptor for the function

bInterfaceProtocol. A protocol code for the function shared by the associated interfaces. For classes that don't specify a value to use, the preferred value for existing device classes is the bInterfaceProtocol value from the descriptor of the first associated interface.

iInterface. Index to a string that describes the function. This value is zero if there is no string descriptor.

Interface Descriptor

The interface descriptor provides information about a function or feature that a device implements. The descriptor contains class, subclass, and protocol information and the number of endpoints the interface uses.

A configuration can have multiple interfaces that are active at the same time. The interfaces may be associated with a single function or they may be unrelated. A configuration can also support alternate, mutually exclusive interfaces. The host can request an alternate interface with a Set_Interface request and read the current interface number with a Get_Interface request. Each interface has its own interface descriptor and subordinate descriptors. Devices that use isochronous transfers must have alternate interfaces because the default interface must request no isochronous bandwidth. Changing interfaces is simpler than changing configurations.

A host retrieves interface descriptors by requesting the configuration descriptor for the configuration the interface belongs to.

An interface descriptor has nine fields. Table 4-8 lists the fields in the order they occur in the descriptor. Many devices don't use the values in all of the fields, such as those that enable alternate settings and protocols. The following descriptions group the information by function.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant INTERFACE (04h).

The Interface

iInterface. Index to a string that describes the interface. This value is zero if there is no string descriptor.

bInterfaceNumber. Identifies the interface. In a composite device, a configuration has multiple interfaces that are active at the same time. Each interface must have a descriptor with a unique value in this field. The default is zero.

bAlternateSetting. When a configuration supports multiple, mutually exclusive interfaces, each of the interfaces has a descriptor with the same value in bInterfaceNumber and a unique value in bAlternateSetting. The Get_Interface request retrieves the currently active setting. The Set_Interface request selects the setting to use. The default is zero.

bNumEndpoints. The number of endpoints the interface supports in addition to Endpoint 0. For a device that supports only Endpoint 0, NumEndpoints is zero.

bInterfaceClass. Similar to bDeviceClass in the device descriptor, but for devices with a class specified by the interface. Table 4-9 shows defined codes. Values from 01h to FEh are reserved for USB-defined classes. FFh indicates a vendor-defined class. Zero is reserved.

bInterfaceSubClass. Similar to bDeviceSubClass in the device descriptor, but for devices with a class defined by the interface. For interfaces that

Table 4-8: The interface descriptor has nine fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Interface (04h)
2	bInterfaceNumber	1	Number identifying this interface
3	bAlternateSetting	1	Value used to select an alternate setting
4	bNumEndpoints	1	Number of endpoints supported, not counting Endpoint 0
5	bInterfaceClass	1	Class code
6	bInterfaceSubclass	1	Subclass code
7	bInterfaceProtocol	1	Protocol code
8	iInterface	1	Index of string descriptor for the interface

belong to a class, this field may specify a subclass within the class. If bInterfaceClass is zero, bInterfaceSubclass must be zero. If bInterfaceClass is between 01h and FEh, bInterfaceSubclass must zero or a code defined by a USB specification. A value of FFh means that the subclass is specific to the vendor. The diagnostic-device, wireless-controller, and application-specific classes have defined subclasses.

bInterfaceProtocol. Similar to bDeviceProtocol in the device descriptor, but for devices whose class is defined by the interface. May specify a protocol defined by the selected bInterfaceClass or bInterfaceSubClass. If bInterfaceClass is between 01h and FEh, bInterfaceProtocol must zero or a code defined by a USB specification.

Endpoint Descriptor

Each endpoint specified in an interface descriptor has an endpoint descriptor. Endpoint 0 never has a descriptor because every device must support Endpoint 0, the device descriptor contains the maximum packet size, and the USB specification defines everything else about the endpoint. A host retrieves endpoint descriptors by requesting the configuration descriptor for the configuration the endpoints belong to.

Table 4-9: The bInterfaceClass field in the interface descriptor can name a class the interface belongs to.

Class Code (hexadecimal)	Description
01	Audio
02	(Communication Device Class) Communication Interface
03	Human Interface Device
05	Physical
06	Image
07	Printer
08	Mass storage
09	Hub
0A	(Communication Device Class) Data Interface
0B	Smart Card
0D	Content Security
0E	Video
DC	Diagnostic device (can also be declared at the device level) bInterfaceSubClass = 1 for Reprogrammable Diagnostic Device with bInterfaceProtocol = 1 for USB2 Compliance Device
E0	Wireless controller (can also be declared at device level) bInterfaceSubClass = 1 for RF Controller with bInterfaceProtocol = 1 for Bluetooth Programming Interface
FE	Application specific bInterfaceSubClass = 1 for Device Firmware Update bInterfaceSubClass = 2 for IrDA Bridge bInterfaceSubClass = 3 for Test and Measurement
FF	Vendor specific (can also be declared at the device level)

Table 4-10 lists the endpoint descriptor's six fields in the order they occur in the descriptor. The following descriptions group the information by function.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant ENDPOINT (05h).

Table 4-10: The endpoint descriptor has six fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Endpoint (05h)
2	bEndpointAddress	1	Endpoint number and direction
3	bmAttributes	1	Transfer type supported
4	wMaxPacketSize	2	Maximum packet size supported
5	bInterval	1	Maximum latency/polling interval/NAK rate

The Endpoint

bEndpointAddress. Contains the endpoint number and direction. Bits 0 through 3 are the endpoint number. Low-speed devices can have a maximum of 3 endpoints (usually numbered 0 through 2), while full- and high-speed devices can have 16 (0 through 15). Bit 7 is the direction: Out = 0, In = 1, Bidirectional (for control transfers) = ignored. Bits 4, 5, and 6 are unused and must be zero.

bmAttributes. Bits 1 and 0 specify the type of transfer the endpoint supports. 00=Control, 01=Isochronous, 10=Bulk, 11=Interrupt. For Endpoint 0, Control is assumed.

In USB 1.1, bits 2 through 7 were reserved. USB 2.0 uses bits 2 through 5 for full- and high-speed isochronous endpoints. Bits 3 and 2 indicate a synchronization type: 00=no synchronization, 01=asynchronous, 10=adaptive, 11=synchronous. Bits 5 and 4 indicate a usage type: 00=data endpoint, 01=feedback endpoint, 10=implicit feedback data endpoint, 11=reserved. For non-isochronous endpoints, bits 2 through 5 must be zero. For all endpoints, bits 6 and 7 must be zero.

wMaxPacketSize. The maximum number of data bytes the endpoint can transfer in a transaction. The allowed values vary with the device speed and type of transfer.

Bits 10 through 0 are the maximum packet size, from 0 to 1024 (0 to 1023 in USB 1.x). In USB 2.0, bits 12 and 11 indicate how many additional transactions per microframe a high-speed endpoint supports: 00=no addi-

tional transactions (total of 1 transaction per microframe), 01=1 additional (total of 2 transactions per microframe), 10=2 additional (total of 3 transactions per microframe), 11=reserved. In USB 1.x, these bits were reserved and set to zero. Bits 13 through 15 are reserved and must be zero.

bInterval. Can indicate the maximum latency for polling interrupt endpoints, the interval for polling isochronous endpoints, or the maximum NAK rate for high-speed bulk OUT or control endpoints. The allowed range and how the value is used varies with the device speed, the transfer type, and whether or not the device complies with USB 2.0.

For low-speed interrupt endpoints, the maximum latency equals bInterval in milliseconds. The value may range from 10 to 255.

For all full-speed interrupt endpoints and for full-speed isochronous endpoints on 1.x devices, the interval equals bInterval in milliseconds. For interrupt endpoints, the value may range from 1 to 255. For isochronous endpoints in 1.x devices, the value must be 1. For isochronous endpoints in full-speed 2.0 devices, values from 1 to 16 are allowed, and the interval is calculated as $2^{\text{bInterval-1}}$, allowing a range from 1 millisecond to 32.768 seconds.

For full-speed bulk and control transfers, the value is ignored.

For high-speed endpoints, the value is in units of 125 microseconds, which is the width of a microframe. The value for interrupt and isochronous endpoints may range from 1 to 16, and the interval is calculated as $2^{\text{bInterval-1}}$ to allow a range from 125 microseconds to 4.096 seconds.

For high-speed bulk OUT and control endpoints, the value indicates the endpoint's maximum NAK rate. This value is relevant when the device has received data and returned ACK, and the host has more data to send in the transfer. By returning ACK, the device is saying that it expects to be able to accept the next transaction's data. (Otherwise the device would return NYET.) If the next data packet arrives and for some reason the device can't accept the packet, the endpoint returns NAK. The bInterval value says that the endpoint will return NAK no more than once in each period specified by bInterval. The value can range from 0 to 255 microframes. A value of

zero means the endpoint will never NAK. The host isn't required to use the maximum-NAK-rate information.

String Descriptor

A string descriptor contains descriptive text. The USB 2.0 specification defines descriptors that can contain indexes to various strings, including strings that describe the manufacturer, product, serial number, configuration, and interface. Class- and vendor-specific descriptors can contain indexes to additional string descriptors. Support for string descriptors is optional, though a class may require them. A host retrieves a string descriptor by sending a Get_Descriptor request with the high byte of the Setup transaction's wValue field equal to 3. Table 4-11 shows the descriptor's fields and their purposes.

The Descriptor

bLength. The number of bytes in the descriptor.

bDescriptorType. The constant STRING (03h).

The String

When the host requests a String descriptor, the low byte of the wValue field is an index value. An index value of zero has the special function of requesting language IDs, while other index values request strings that may contain any text.

wLANGID[0...n]. Used in string descriptor 0 only. String descriptor 0 contains one or more 16-bit language ID codes that indicate the languages that the strings are available in. The code for English is 0009h, and the subcode for U.S. English is 0004h. These are likely to be the only codes supported by an operating system. The wLANGID value must be valid for any of the other strings to be valid. Devices that return no string descriptors must not return an array of language IDs. The USB-IF's web site has a list of defined USB language IDs.

bString. For values 1 and higher, the String field contains a Unicode string. Unicode uses 16 bits to represent each character. With a few exceptions,

Table 4-11: A string descriptor has three or more fields.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant String (03h)
2	bSTRING or wLANGID	varies	For string descriptor 0, an array of 1 or more Language Identifier codes. For other string descriptors, a Unicode string.

ANSI character codes 00h through 7Fh correspond to Unicode values 0000h through 007Fh. For example, a product string for a product called "Gizmo" would contain five 16-bit Unicode values that represent the characters in the product name: 0047 0069 007A 006D 006F. The strings are not null-terminated.

Other Standard Descriptors

The USB 2.0 specification lists three additional descriptor codes for interface_power, OTG, and debug descriptors.

The interface_power descriptor is defined in a proposed Interface Power Management specification to enable interfaces to manage their power consumption individually. The specification was proposed by Microsoft in 1998 but hasn't been approved or implemented. The document describing this descriptor's structure and use is *USB Feature Specification: Interface Power Management*.

The OTG descriptor is required for devices that support On-The-Go's Host Negotiation Protocol (HNP) or Session Request Protocol (SRP). The descriptor indicates the supported protocols. Chapter 20 has more about this descriptor.

The debug descriptor is defined in a proposed specification for USB2 Debug Devices. A debug device connects to the optional debug port defined in the EHCI specification for high-speed host controllers. The debug port and device are intended to replace the RS-232 port that PCs have long used for debugging purposes.

The Microsoft OS Descriptor

Microsoft has defined its own Microsoft OS descriptor for use with devices in vendor-defined classes. The descriptor is intended to assist in providing Windows-specific data such as icons and registry settings.

The descriptor consists of a special String descriptor and one or more Microsoft OS feature descriptors. The String descriptor must have an index of EEh and contains an embedded signature. Windows XP SP1 and later request this string descriptor on first attachment. A device that doesn't support this descriptor should return a STALL.

If a device contains a Microsoft OS String descriptor, Windows requests additional Microsoft-specific descriptors. Future editions of the Windows DDK will have more documentation about these descriptors.

Descriptors in 2.0-compliant Devices

If you're upgrading a 1.x-complaint device to 2.0, what changes are required in the descriptors? In a dual-speed device, can you detect whether a device is using full or high speed by reading its descriptors? This section answers these questions.

Making 1.x Descriptors 2.0-compliant

Table 4-12 lists the descriptor fields whose contents may require changes to enable a 1.x device to comply with the USB 2.0 specification. For all except some devices that have isochronous endpoints, the one and only required change is this: in the device descriptor, the bcdUSB field must be 0200h.

As Chapter 3 explained, a USB 2.0 device's default interface(s) must request no isochronous bandwidth. And because the default interface is of no use for transferring isochronous data, a device that wants to do isochronous transfers must support at least one alternate interface setting, and the alternate interface descriptor will have at least one subordinate endpoint descriptor. Some 1.x devices meet this requirement already.

Table 4-12: The descriptors in a 1.x-compliant device require very few changes to comply with USB 2.0.

Descriptor	Field	Change
Device	bcdUSB	Set to 0200h.
Endpoint	wMaxPacketSize	Isochronous only: must be 0 in the default configuration.

The USB 2.0 specification also adds two new descriptors and functions for bits in existing fields, but the new descriptors are used only in dual-speed devices and the other descriptors are backwards compatible with 1.x.

Full-speed isochronous endpoints have a few new, optional abilities. The endpoint descriptor can specify synchronization and usage types (bmAttributes field), and the interval can be greater than 1 millisecond (bInterval field). In 1.x descriptors, these bits default to 0 (no synchronization) and 1 (one millisecond).

When selecting bInterval values for interrupt and isochronous endpoints, don't forget that the relation between bInterval and the interval time will vary depending on the transfer type and speed. For low- and full-speed interrupt endpoints, the interval equals bInterval in milliseconds. For full-speed isochronous endpoints, the interval equals $2^{bInterval-1}$ in milliseconds. For high-speed interrupt and isochronous endpoints, the interval equals $2^{bInterval-1}$ in units of 125 microseconds. Note that if bInterval = 1, the full-speed interval is 1 millisecond in both USB 1.x and USB 2.0. So a 1.x isochronous endpoint, which must have bInterval = 1, requires no changes to comply with USB 2.0.

If you upgrade a full-speed device to support high speed as well, the device needs a device_qualifier descriptor, an other_speed_configuration descriptor, and a set of descriptors for the high-speed configuration. Any interrupt endpoints in the default interface must have a maximum packet size of 64 or less. A USB 2.0 device that supports only low speed or only full speed must return STALL in response to requests for the device_qualifier and other_speed_configuration descriptors.

Detecting the Speed of a Dual-Speed Device

A high-speed device must respond to enumeration requests at full speed, and the device may also be completely functional at full speed. As Chapter 1 explained, a high-speed-capable device must use full speed if it has a 1.x host or if there is a 1.x hub between the host and device. Applications and device drivers normally don't need to know which speed a dual-speed device is using because all of the speed-related details are handled at a lower level. Windows provides no straightforward way to learn a device's speed. But if a host application wants to know, there are a few techniques that can detect a the bus speed for many devices.

If a device has a bulk endpoint, you can learn the current speed by examining the endpoint descriptor in the active configuration. The wMaxPacketSize field must be 512 in a high-speed device and can't be 512 in a full-speed device. If there is no bulk endpoint, the wMaxPacketSize of an interrupt or isochronous endpoint provides speed information if the endpoint uses a maximum packet size available only at high speed. For an interrupt endpoint, a wMaxPacketSize greater than 64 indicates high speed. If the wMaxPacketSize is 64 or less, the device may be using full or high speed. For isochronous endpoints, a wMaxPacketSize of 1024 indicates high speed. If wMaxPacketSize is 1023 or less, the device may be using full or high speed.

If you're writing the device firmware, you can provide speed information in the optional strings indexed by the configuration and other_speed_configuration descriptors. For example, the string indexed by the configuration descriptor might contain the text "high speed," and the string indexed by the other_speed_configuration descriptor might contain the text "full speed." Applications can then read the configuration string to learn the current speed.

The USBView application in the Windows DDK shows how applications can read descriptors.