7

Device Classes

This chapter is an introduction to the defined USB classes including how to decide whether a new design will fit a defined class or will require a custom driver.

About Classes

Most USB devices have much in common with other devices that perform similar functions. All mice send information about mouse movements and button clicks. All drives transfer files. All printers receive data to print and send status information back to the host.

When a group of devices or interfaces share many attributes or provide or request similar services, it makes sense to define the attributes and services in a class specification. The specification can serve as a guide for developers who design and program devices in the class and for programmers who write device drivers for host systems that communicate with the devices. Operating systems can provide drivers for common classes, eliminating the need for device vendors to provide drivers for devices in those classes.

When a device in a supported class has unique features or abilities not included in a class's driver, a device vendor sometimes can provide a filter driver to support the added features and abilities, rather than writing a complete device driver. In other cases, a filter driver isn't feasible and the device requires a custom driver.

Even if a device's class isn't supported by the operating system, a class may be supported in a future edition of the operating system. Firmware that complies with a class specification is likely to be compatible with any driver added in future editions of the operating system.

Device Working Groups

The USB-IF releases class specifications developed by Device Working Groups, whose members have expertise and interest in a particular area. A special case is the hub class, which is defined in the main USB 2.0 specification rather than in a separate document. Every operating system must support the hub class because the host requires a root hub to do any communications.

The defined classes cover most common device functions. A specification with a version number of 1.0 or higher is an approved specification and is suitable for use as a reference in developing devices and drivers for commercial release. Table 7-1 lists the classes with approved specifications.

Windows includes drivers to support many of these classes. As Windows and the class specifications have evolved, the number of supported classes and the level of support for the classes have improved. For some of the more obscure classes, such as Device Firmware Upgrade, Windows doesn't provide a driver even though the specification was approved years ago. Chapter 4 listed the defined class codes devices may have in their device and interface descriptors. Table 7-2 shows the class drivers added in each edition of Windows.

Elements of a Class Specification

All USB class specifications are based on the Common Class specification, which describes what information a class specification should contain and

Table 7-1: These classes have approved class specifications.

Class	Descriptor Where Class Is Declared
Audio	Interface
Chip/Smart Card Interface	Interface
Communication	Device or Interface
Content Security	Interface
Device Firmware Upgrade	Interface (subclass of Application Specific Interface)
Human Interface (HID)	Interface
IrDA Bridge	Interface (subclass of Application Specific Interface)
Mass Storage	Interface
Printer	Interface
Still Image Capture	Interface
Test and Measurement	Interface (subclass of Application Specific Interface)
Video	Interface

how to organize a specification document. A class specification defines the number and type of required and optional endpoints devices in a class may have. A specification may also define or name formats for data to be transferred, including both application data (such as keypresses or video data) and status and control information relating to the device and its operation. Some class specifications define functions or capabilities that describe how the data being transferred will be used. For example, the HID class has Usage Tables that define how to interpret data sent by keyboards, mice, joysticks, and other HIDs. Some classes use USB to transfer data in a format defined by another specification. An example is the SCSI commands used by mass-storage devices.

A class specification may define values for items in standard descriptors as well as defining class-specific descriptors, interfaces, endpoint uses, and control requests. For example, the device descriptor for a hub includes a bDeviceClass value of 09h to indicate that the device belongs to the hub class. The hub must have a class-specific hub descriptor with a descriptor type of 29h. Hubs must also support class-specific requests. For example, when the host sends a Get_Port_Status request to a hub with a port number in the Index field, the hub responds with status information for the port. A

Table 7-2: Microsoft has added USB class support with each release of Windows. The releases are listed top to bottom from earliest to latest. Except as noted, each release also includes the drivers provided with earlier releases.

Windows Edition	USB Version Compliance	USB Drivers Added
Windows 98 Gold (original release)	1.0	Audio
		HID 1.0
		Video (USB camera minidriver library USBCAMD 1.0; not supported under Windows 2000)
Windows 98 SE	1.1	Communication device: modem
		HID 1.1 (adds the ability to do interrupt OUT transfers)
		Still image (first phase/preliminary)
Windows 2000	1.1 (2.0 support added in Service Pack 4 (SP4)	Mass storage. Support for multiple LUNs (partitions) added in Service Pack 3 (SP3).
		Printer
		Communication device: Remote NDIS (Network Device Interface Specification)
		Still image (much improved)
		Chip/Smart Card Interface (available from Windows update)
Windows Me	1.1	Audio: MIDI
		Video (USB camera minidriver library USBCAMD 2.0)
Windows XP	1.1; 2.0 support added in Service Pack 1 (SP1); interface association descriptor support added in Service Pack 2 (SP2).	Audio: MIDI, improved
		Video-class driver added in Service Pack 2 (SP2).

class may also require a device to support specific endpoints or comply with tighter timing for standard requests.

Defined Classes

The following sections introduce the defined classes. I don't attempt to repeat every detail in the specification documents. Instead, my goal is to give enough information to help you decide what class a new design might fit into, what resources a device requires to support a class's communications, examples of device controllers to use for devices in the class, and what level of support, if any, to expect for the class under Windows.

Audio

The audio class is for devices that send or receive audio data, which may contain encoded voice, music, or other sounds. Audio functions are often part of a device that also supports video, storage, or other functions. Audio devices can use isochronous transfers for audio streams or bulk transfers for data encoded using the MIDI (Musical Instrument Digital Interface) protocol.

This section describes version 1.0 of the audio specification. At this writing, version 2.0 is under development. Version 2.0 will not be backwards compatible with version 1.0. In other words, a 2.0 device won't work with a 1.0 host driver. The proposed changes in version 2.0 include complete support for high-speed operation, use of the interface association descriptor, and support for many new capabilities and controls.

Documentation

The audio specification has separate Device Class Definition documents for Audio Devices, Audio Data Formats, Terminal Types, and MIDI Devices. At this writing, the latest version of each of these is 1.0. The MIDI standard is available from the MIDI Manufacturers Association at *www.midi.org*.

Overview

Each audio function in a device has an Audio Interface Collection that consists of one or more interfaces. The interfaces include one AudioControl (AC) interface, zero or more AudioStreaming (AS) interfaces and zero or more MIDIStreaming (MS) interfaces (Figure 7-1). In other words, every

Audio Interface Collection has an AudioControl interface, while Audio-Streaming and MIDIStreaming interfaces are optional.

An AudioControl interface can enable accessing controls such as volume, mute, bass, and treble. An AudioStreaming interface transfers audio data in isochronous transfers. A MIDIStreaming interface transfers MIDI data. MIDI is a standard for controlling synthesizers, sound cards, and other electronic devices that generate music and other sounds. A MIDI representation of a sound includes values for pitch, length, volume, and other characteristics. A pure MIDI hardware interface carries asynchronous data at 31.25 kilobits/sec. A USB interface that carries MIDI data uses the MIDI data format but doesn't use MIDI's asynchronous interface. Instead, the MIDI data travels on the bus in bulk transfers.

A device can have multiple Audio Interface Collections that are active at the same time, with each collection controlling an independent audio function.

Descriptors

Each audio interface type uses standard and class-specific descriptors to enable the host to learn about the interface, its endpoints, and what kinds of data the endpoints expect to transfer.

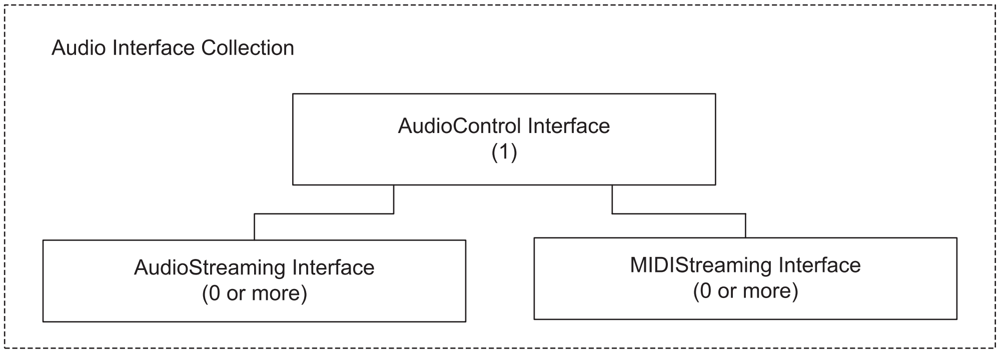

Figure 7-1: Each audio function has an Audio Interface Collection that contains one or more interfaces.

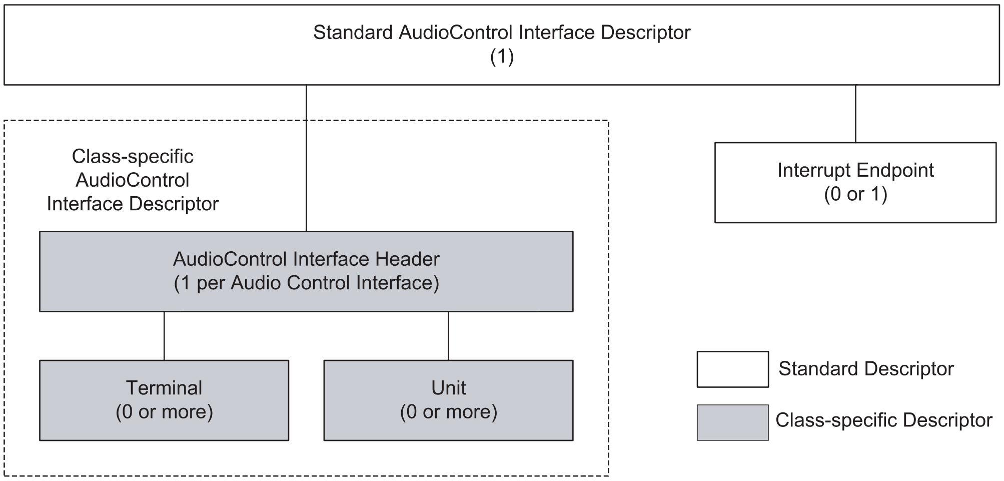
Standard AudioControl Interface Descriptor
(1)
Class-specific AudioControl Interface Descriptor
Interrupt Endpoint
(0 or 1)
AudioControl Interface Header
(1 per Audio Control Interface)
Terminal
(0 or more)
Unit
(0 or more)
Standard Descriptor
Class-specific Descriptor

Figure 7-2: An AudioControl interface contains descriptors for audio Terminals and Units.

The AudioControl Interface. Figure 7-2 shows the descriptors in an AudioControl interface. In the AudioControl interface descriptor, bInterfaceClass = 01h to identify the interface as audio class and bInterfaceSubclass = 01h to identify the subclass as AudioControl.

Following the AudioControl interface descriptor is a Class-specific AC interface descriptor, which consists of a series of descriptors for the AudioControl interface. The Class-specific AC Interface Header descriptor contains the total length of itself and all of the Terminal and/or Unit descriptors in the interface.

A Terminal descriptor contains information about an addressable logical object that represents a USB endpoint or other interface to the outside world. Every IN or OUT isochronous endpoint in an audio interface must have an associated Output or Input Terminal with a Terminal descriptor. The audio function receives audio information from the host at an Input Terminal and transmits audio information to the host at an Output Terminal. Note that the terms Input Terminal and Output Terminal are from the perspective of the audio function, while USB endpoints are named from the perspective of the host. So an IN endpoint has an associated Output Terminal, while an OUT endpoint has an associated Input Terminal. Other Terminals in a function can represent interfaces to audio components such as microphones and speakers.

An audio function might receive a microphone's output at an Input Terminal and transmit the received audio data to the host at an Output Terminal that represents a USB IN endpoint. Or an audio function might receive audio data from the host at an Input Terminal that represents an OUT endpoint and send the received data to a speaker at an Output Terminal.

A Unit descriptor contains information about an addressable logical object that represents a subfunction within an audio function. Table 7-3 shows the Unit types defined in the specification.

If the AudioControl Interface has an interrupt endpoint, the interface includes an endpoint descriptor for the endpoint.

The AudioStreaming Interface. Following the AudioControl interface descriptor, an Audio Interface Collection may have one or more AudioStreaming interface descriptors with bInterfaceClass field = 01h to identify the interface as audio class and bInterfaceSubclass = 02h to identify the subclass as AudioStreaming. Figure 7-3 shows the descriptors.

Following each AudioStreaming interface descriptor is a class-specific AudioStreaming interface descriptor, which identifies the Terminal associated with the interface and contains information about the data format and any delay the function requires for internal processing. The *Audio Data Formats* specification lists the supported formats, which include Pulse Code Modulation (PCM), Digital Audio Compression (AC-3), and MPEG. A class-specific AS Format Type descriptor contains more information about the format. Some formats require an additional AS Format-specific Type descriptor.

Each AudioStreaming interface can have one isochronous endpoint. The endpoint has a standard endpoint descriptor and a class-specific AS Isochronous Audio Data endpoint descriptor. The class-specific descriptor indicates which audio controls the endpoint supports, specifies whether the endpoint requires all except zero-length data packets to contain wMaxPacketSize bytes, and can provide synchronizing information. Some endpoints also have a class-specific AS Isochronous Synch endpoint descriptor.

The MIDIStreaming Interface. To support MIDI data, an Audio Interface Collection can have one or more MIDIStreaming interfaces. Figure 7-4 shows the descriptors. In the MIDIStreaming interface descriptor, bInterfaceClass = 01h to identify the interface as audio class and bInterfaceSubclass = 03h to identify the subclass as MIDIStreaming.

Following this descriptor is a class-specific MIDIStreaming (MS) interface descriptor that consists of a series of descriptors for the interface. The first descriptor in the series is the class-specific MS Interface Header descriptor, which contains the total length of itself plus all of the Jack and/or Element descriptors that follow.

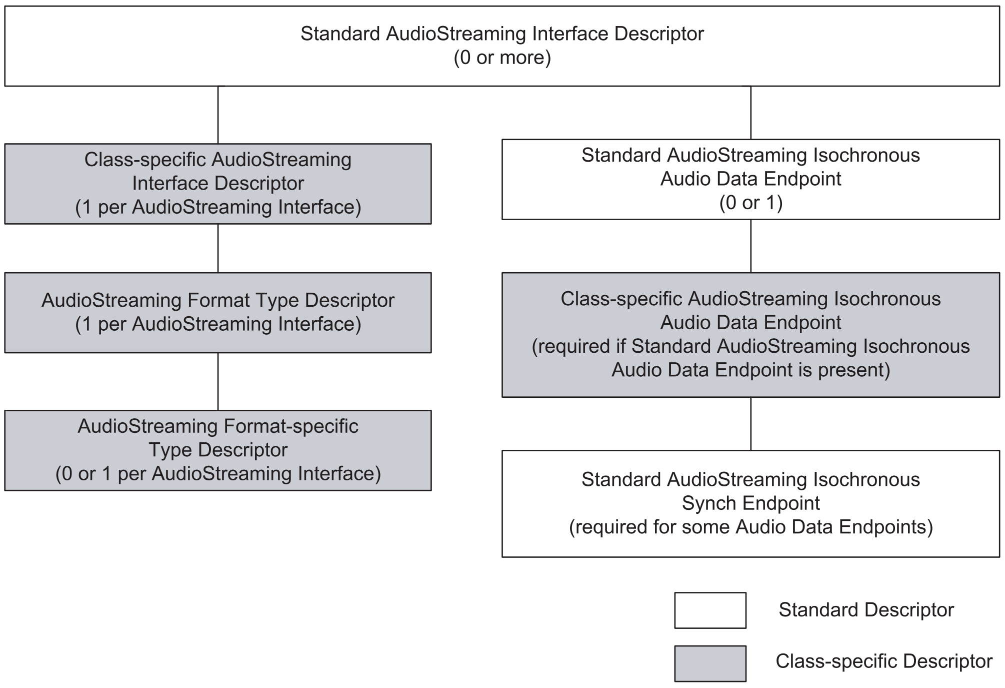
Standard AudioStreaming Interface Descriptor
(0 or more)
Class-specific AudioStreaming Interface Descriptor
(1 per AudioStreaming Interface)
AudioStreaming Format Type Descriptor
(1 per AudioStreaming Interface)
AudioStreaming Format-specific Type Descriptor
(0 or 1 per AudioStreaming Interface)
Standard AudioStreaming Isochronous Audio Data Endpoint
(0 or 1)
Class-specific AudioStreaming Isochronous Audio Data Endpoint
(required if Standard AudioStreaming Isochronous Audio Data Endpoint is present)
Standard AudioStreaming Isochronous Synch Endpoint
(required for some Audio Data Endpoints)
Standard Descriptor
Class-specific Descriptor

Figure 7-3: An AudioStreaming interface contains descriptors for an isochronous endpoint that carries audio data.

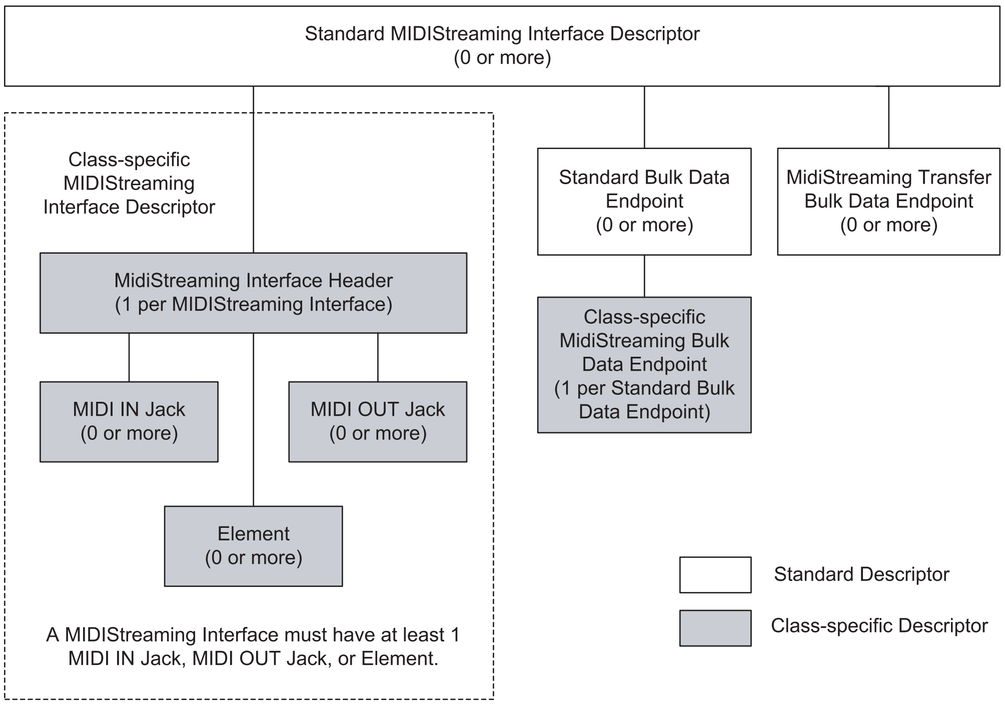
Standard MIDIStreaming Interface Descriptor
(0 or more)
Class-specific MIDIStreaming Interface Descriptor
MidiStreaming Interface Header
(1 per MIDIStreaming Interface)
MIDI IN Jack
(0 or more)
MIDI OUT Jack
(0 or more)
Element
(0 or more)
A MIDIStreaming Interface must have at least 1 MIDI IN Jack, MIDI OUT Jack, or Element.
Standard Bulk Data Endpoint
(0 or more)
MidiStreaming Transfer Bulk Data Endpoint
(0 or more)
Class-specific MidiStreaming Bulk Data Endpoint
(1 per Standard Bulk Data Endpoint)
Standard Descriptor
Class-specific Descriptor

Figure 7-4: A MIDIStreaming interface contains descriptors for bulk endpoints that carry MIDI data.

An Element converts between MIDI and audio data streams or other MIDI streams. A MIDI IN Jack receives data from the outside world, and a MIDI OUT Jack provides data to the outside world. An Embedded MIDI Jack is a jack that represents a USB endpoint. An Embedded MIDI OUT Jack represents an IN endpoint, and an Embedded MIDI IN Jack represents an OUT endpoint. (The MIDI OUT and MIDI IN names are from the perspective of the MIDI function, while the IN endpoint and OUT endpoint names are from the perspective of the USB host.) An External MIDI Jack is a physical jack that connects to a MIDI device.

Every USB MIDI device contains a USB/MIDI converter that converts between USB data and the data at the Embedded MIDI Jack(s). Each USB endpoint can connect to up to 16 Embedded MIDI Jacks. Data travels in 32-bit USBMIDI Event Packets, with the first 4 bits designating a specific

Table 7-3: The specification for the audio class defines these Unit types.

Unit Type	Description
Mixer	Transforms a number of logical input channels into a number of logical output channels.
Selector	Selects from *n* audio channel clusters and routes them unaltered to a single output audio channel cluster
Feature	Provides controls such as volume, tone control, and mute
Processing	Transforms multiple logical input channels into a single audio channel cluster
Up/Down-mix Processing	Provides facilities to derive *m* output audio channels from *n* input audio channels.
Dolby Prologic Processing	Extracts additional audio data (a specialized derivative of the Up/Down-mix Processing Unit)
3D-Stereo Extender Processing	Processes an existing stereo sound track to add a spaciousness effect.
Reverberation Processing	Adds room-acoustics effects.
Chorus Processing	Adds chorus effects.
Dynamic Range Compressor	Intelligently limits the dynamic range.
Extension Unit	Enables adding vendor-specific building blocks.

Embedded MIDI Jack. Inside a device, External Jacks connect to Embedded MIDI Jacks, other External MIDI Jacks, or Elements. Multiple External MIDI OUT Jacks can implement MIDI PARALLEL OUT. Each Element, MIDI IN Jack, and MIDI OUT Jack has a class-specific descriptor.

A MIDIStreaming interface can have one or more Standard MS Bulk Data endpoints and one or more Class-specific MS Transfer Bulk Data endpoints. Many MIDI interfaces handle all traffic with one Standard Bulk Data IN endpoint and one Standard Bulk Data OUT endpoint. For audio streams that require more bandwidth, an interface can have one or more MS Transfer Bulk Data endpoints. A host can use the class-specific Set_Endpoint_Control request to dynamically allocate a Transfer Bulk Data endpoint to an Element. Typical applications for a Transfer Bulk Data endpoint are transferring DownLoadable Sounds (DLS) to a Synthesizer Element and transferring program code to a programmable Element.

Each endpoint of either type has a standard endpoint descriptor. Each Standard Bulk Data endpoint also has an MS Bulk Data endpoint descriptor that names the Embedded MIDI Jacks associated with the endpoint.

Class-specific Requests

The audio class defines optional class-specific requests for setting and getting the state of audio controls, accessing memory, and requesting status information.

Chips

USB-capable chips are available with built-in support for audio functions. The support includes codec (compressor/decompressor) functions, analog-to-digital converters (ADCs), digital-to-analog converters (DACs), and support for Sony/Philips Digital Interface (S/PDIF) encoding for transmitting audio data in digital format.

Texas Instruments' PCM2900 is a stereo audio codec with a full-speed USB port and 16-bit ADC and DAC. The chip has an AudioControl interface, an AudioStreaming interface for each direction, and a HID interface that reports the status of three pins on the chip. The chip requires no user programming but has the option to use a vendor-specific Vendor ID, Product ID, and strings. The PCM2902 adds support for S/PDIF encoding. Another option for a USB codec is Philips Semiconductors UDA1325.

Texas Instruments' PCM2702 is a 16-bit stereo DAC with a full-speed USB interface. The chip can accept data sampled at 48, 44.1, and 32 kilohertz using either 16-bit stereo or monaural audio data. The chip supports digital attenuation and soft-mute features.

Texas Instruments' TUSB3200A USB Streaming Controller contains an 8052-compatible microcontroller that supports up to seven IN endpoints and seven OUT endpoints. The audio support includes a codec port interface, a DMA controller with four channels for streaming isochronous data packets to and from the codec port, and a phase lock loop (PLL) and adaptive clock generator (ACG) to support synchronization modes.

Windows Support

Under Windows, the *usbaudio.sys* minidriver supports USB audio devices, including MIDI devices. In Windows editions up to and including Windows XP, the driver supports a subset of the features in the USB audio specification. Microsoft's Universal Audio Architecture (UAA) initiative promises an improved driver architecture for future Windows editions. *USB Audio Devices and Windows* is a white paper from Microsoft that details the abilities and limits of Windows XP's audio driver.

Applications can access USB audio devices using the DirectMusic and DirectSound components of Windows' DirectX technology or using Windows Multimedia audio functions.

Chip/Smart Card Interface

Smart cards are the familiar plastic cards used for phone calls, gift cards, keyless entry, access to toll roads and mass transit, storing medical and insurance data, enabling of satellite TV receivers, and other applications that require storing small-to-moderate quantities of information with easy and portable access.

Each card contains a module with memory and often a CPU. Many cards allow updating of their contents, to change a monetary value or an entry code, for example. Some cards have exposed electrical contacts, while others communicate via embedded antennas. Another term for smart card is chip card.

To access a smart card, you connect it to a Chip Card Interface Device (CCID), typically by inserting the card into a slot or waving a contactless card by a reader. A popular term for CCID is smart-card reader, though many CCIDs can also write to cards. USB enters the picture because some CCIDs have USB interfaces for communicating with USB hosts.

Documentation

The specification *USB Chip/Smart Card Interface Devices* defines a protocol for CCIDs with USB interfaces. The current version at this writing is 1.0. The ISO/IEC 7816 standard (available from *www.iso.ch*) defines the physical and electrical characteristics and commands for communicating with smart cards.

Overview

Every CCID must have a bulk endpoint in each direction. All readers with removable cards must also have an interrupt IN endpoint.

The host and device exchange messages on the bulk pipes. A CCID message consists of a 10-byte header followed by message-specific data. The specification defines 14 commands that the host can use to send data and status and control information in messages. Every command requires at least one response message from the CCID. A response contains a message code and status information and may contain additional requested data. The device uses the interrupt endpoint to report errors and the insertion or removal of a card.

Descriptors

A CCIS function is defined at the interface level. In the interface descriptor, bInterfaceClass = 0Bh to indicate the CCID class. Following the interface descriptor is a class-specific CCID Class descriptor with bDescriptorType = 21h. The class descriptor contains parameters such as the number of slots, slot voltages, supported protocols, supported clock frequencies and data rates, and maximum message length.

Class-specific Requests

There are three class-specific control requests:

Request	bRequest	Required?
Abort	01h	yes
Get_Clock_Frequencies	02h	yes, if the CCID doesn't support automatic selection of clock frequency (as specified in the CCID class descriptor, dwFeatures, bit 10h)
Get_Data_Rates	03h	yes, if the CCID doesn't support automatic selection of clock frequency (as specified in the CCID class descriptor, dwFeatures, bit 20h)

Chips

A CCID can use just about any full- or high-speed device controller. Some controllers have support for CCID functions built in. Alcor Micro Corporation has the AU9510 CCID chip with a USB interface. Winbond Electronics Corporation's W81E381D is an 8052-compatible microcontroller with USB and smart-card-reader interfaces.

Windows Support

A Windows USB driver for communicating with CCIDs wasn't included with Windows editions up to and including Windwos XP, but a driver is available for Windows 2000 and later via Windows update. Applications use DeviceIoControl API functions to communicate with CCIDs. The driver doesn't support PIN entry or multi-slot readers.

Communication Devices: Modems and Networks

The communication-device class encompasses two broad device types: telephones and "medium-speed" networking devices. Telephones include analog phones and modems, ISDN terminal adapters, and digital phones. Networking devices include ADSL modems, cable modems, and 10BASE-T Ethernet adapters and hubs. The USB interface in a communication device typically carries data that uses application-specific protocols such as V.250

for modem control or Ethernet for local-network traffic. The communication-device class is also an option for other devices accessed via COM-port functions on the host.

Documentation

The main documentation for communication devices is the specification for the communication-device class (CDC). Two subclasses have their own documents. The Wireless Mobile Communications (WMC) subclass includes terminal equipment for wireless devices that can perform multiple functions such as audio and data communications. The Ethernet Emulation Model (EEM) Devices subclass includes devices that send and receive Ethernat frames. At this writing, the latest specification versions are 1.1 for CDC and 1.0 for WMC and EEM. The V.250 standard (formerly known as V.25ter and encompassing the Hayes AT command set) is available from the International Telecommunication Union at *www.itu.int*. The Ethernet standard, IEEE 802.3, is available from *www.ieee.org*.

The Remote Network Driver Interface Specification (NDIS) defines a protocol for using USB and other buses to configure network interfaces and to send and receive Ethernet data. Remote NDIS is based on NDIS, which defines a protocol to manage communications with network adapters and higher-level drivers. NDIS and Remote NDIS are supported by Windows, but not by other operating systems. Documentation for NDIS and Remote NDIS are available from *www.microsoft.com*.

Overview

A communication device is responsible for the tasks of device management, call management (optional), and data transmission. Device management includes controlling and configuring a device and notifying the host of events. Call management involves establishing and terminating telephone calls or other connections. Not all devices require call management. Data transmission is the sending and receiving of application data such as phone conversations or files sent over a modem or network.

The communication device class supports three basic models for communicating. The POTS (Plain Old Telephone Service) model is for communica-

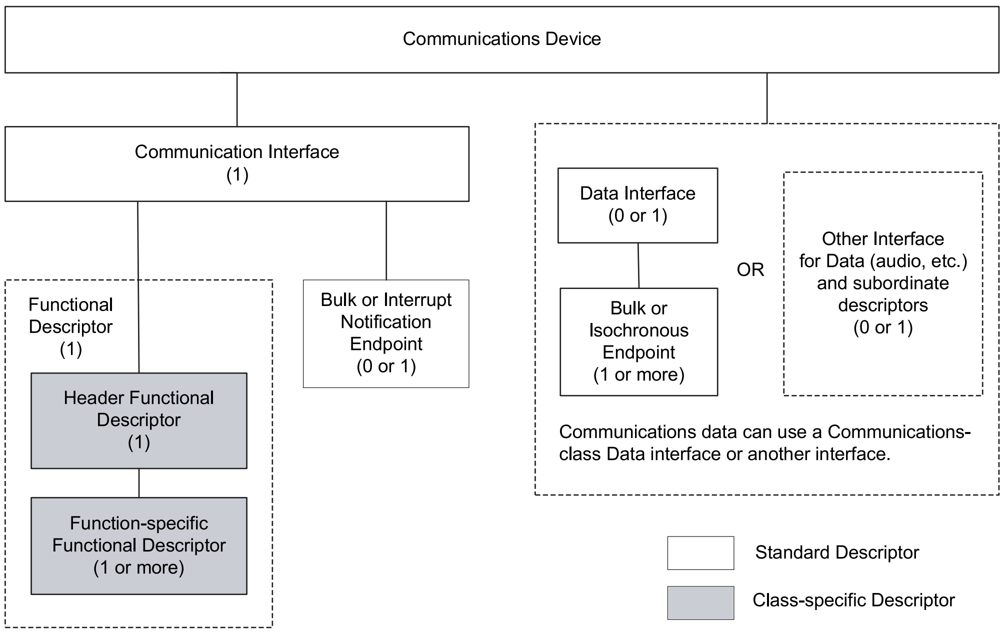
Communications Device
Communication Interface (1)
Functional Descriptor (1)
Header Functional Descriptor (1)
Function-specific Functional Descriptor (1 or more)
Bulk or Interrupt Notification Endpoint (0 or 1)
Data Interface (0 or 1)
Bulk or Isochronous Endpoint (1 or more)
OR
Other Interface for Data (audio, etc.) and subordinate descriptors (0 or 1)
Communications data can use a Communications-class Data interface or another interface.
Standard Descriptor
Class-specific Descriptor

Figure 7-5: A communication-class interface has descriptors for endpoints that carry communication-device class data.

tions via ordinary phone lines. The ISDN model is for communications via phone lines with ISDN interfaces. The Networking model is for communications via Ethernet or ATM (Asynchronous Transfer Mode) networks. Some USB/Ethernet devices use the POTS model with a vendor-specific protocol.

Notifications, which announce events such as ring detect and network connect or disconnect, can travel to the host in an interrupt or bulk pipe. Most devices use interrupt pipes. Each notification consists of an 8-byte header followed by a variable-length data field.

Descriptors

A communication function can be defined at either the device or interface level. If defined at the device level, all of the device's interfaces belong to the communication function. In the device descriptor, bDeviceClass = 02h to indicate the communication-device class (Figure 7-5). If the communication function is defined at the interface level, an associated interface descriptor can specify which interfaces belong to the communication function, assuming that the operating system supports the associated interface descriptor. The 1.1 communication-device class specification doesn't mention the associated-interface descriptor by name but says that a method for associating interfaces is under development and that such a method would be a valid option.

Every communication device must have an interface descriptor with bInterfaceClass = 02h to indicate a Communication interface. This interface handles device management and call management. The bInterfaceSubClass field specifies a communication model. Table 7-4 shows defined values. The bInterfaceProtocol field can name a protocol supported by a subclass. Table 7-5 shows defined values for protocols.

Following the Communication interface descriptor is a class-specific Functional descriptor consisting of a Header Functional descriptor followed by one or more descriptors (also called Functional descriptors) that provide information about a specific communication function. Table 7-6 shows defined values for these descriptors.

One of these descriptors, the Union Functional descriptor, has the special function of defining a relationship among multiple interfaces that form a functional unit. The descriptor designates one interface as the master or controlling interface, which can send and receive certain messages that apply to the entire group. For example, a Communication interface can be a master interface for a group consisting of a Communication interface and a Data interface. The interfaces that make up a group can include communication-class interfaces as well as other related interfaces such as audio and HID.

If the Communication interface has a bulk or interrupt endpoint for event notifications, the endpoint has a standard endpoint descriptor.

A communication device may also have an interface descriptor with bInterfaceClass = 0Ah to indicate a Data interface. A Data interface can have bulk or isochronous endpoints for carrying application data. Each of these endpoints, when present, has a standard endpoint descriptor. Some devices use

Table 7-4: In the interface descriptor for a communication device, the bInterfaceSubClass field indicates the communication model the device supports.

Code	bInterfaceSubClass	Application
00h	RESERVED	–
01h	Direct Line Control Model	Telephone modem with the host providing any data compression and error correction. The device or host may provide modulation/demodulation of the modem data.
02h	Abstract Control Model	Telephone modem with the device providing any data compression, error correction, and modulation/demodulation of the modem data.
03h	Telephone Control Model	Telephone.
04h	Multi-Channel Control Model	ISDN device with multiple, multiplexed channels.
05h	CAPI Control Model	ISDN device with support for COMMON-ISDN-API (CAPI) commands and messages.
06h	Ethernet Networking Control Model	Device that exchanges Ethernet-framed data.
07h	ATM Networking Control Model	ATM device.
08h–0Bh	WMC models	Wireless mobile communications device.
0Ch	Ethernet Emulation Model (EEM)	Device that exchanges Ethernet frames.
0Dh–7Fh	Reserved	Future use.
80h–FEh	Reserved	Vendor specific.

other class or vendor-specific interfaces for data transmission. For example, a telephone might use an audio interface to send and receive voice data.

A USB/Ethernet converter that functions as a Remote NDIS device consists of a Communication interface and a Data interface. In the Communication interface, bInterfaceSubClass = 02h to specify the Abstract Control Model and bInterfaceProtocol = FFh to specify a vendor-specific protocol. (Remote NDIS devices don't use the communication class's Ethernet Control Model.) The Communication interface has an interrupt endpoint. The Data interface has two bulk endpoints. Each endpoint has an endpoint descriptor.

Table 7-5: In the interface descriptor for a communication device, the bInterfaceProtocol field can indicate a protocol the communications model supports.

Code	Description
00h	No class-specific protocol required
01h	AT commands (specified in ITU V.250)
02h–06h	AT commands for WMC devices
07h–FDh	Future use
FEh	External protocol for WMC devices
FFh	Vendor specific

The Ethernet Emulation Model Devices subclass defines an alternate way to use USB to send and receive Ethernet frames. The EEM subclass is intended to be less expensive and more efficient than the Ethernet Networking Control Module subclass defined in the communication-device class specification.

Class-specific Requests

The communication-device class has a variety of class-specific requests for getting and setting status and control information. Not every request is valid for every device. For example, Set_Hook_State requests to place a phone line on or off hook, and Set_Ethernet_Packet_Filter requests to filter Ethernet traffic according to specified settings.

Chips

For modems, Cypress Semiconductor provides several reference designs using EZ-USB controllers and modem components from partner companies.

For USB/Ethernet bridges, Asix Electronics Corporation has several chips, including the AX88172 controller, which converts between full- or high-speed USB and 10- or 100-Mbps Ethernet. The chip's Ethernet interface connects to an external Ethernet PHY. An external serial EEPROM can store the device's Ethernet hardware address, USB descriptors, and configuration data for the converter. The chip has two bulk endpoints for Ethernet

Table 7-6: A Functional descriptor consists of a Header functional descriptor followed by one or more function-specific descriptors.

bInterfaceSubClass	Functional Descriptor Type
00h	Header
01h	Call Management
02h	Abstract Control Management
03h	Direct Line Control Management
04h	Telephone Ringer
05h	Telephone Call and Line State Reporting Capabilities
06h	Union
07h	Country Selection
08h	Telephone Operational Modes
09h	USB Terminal
0Ah	Network Channel Terminal
0Bh	Protocol Unit
0Ch	Extension Unit
0Dh	Multi-channel Management
0Eh	CAPI Control Management
0Fh	Ethernet Networking
10h	ATM Networking
11h–18h	WMC Functional Descriptors
19h–FFh	Reserved

data and an interrupt endpoint for sending notifications. A series of vendor-specific requests enable configuring and reading status information from the chip and accessing three I/O bits.

Kawasaki Microelectronics has several chips that each contain Ethernet, USB, and serial-EEPROM interfaces and a 16-bit CPU. Freescale Semiconductor's 32-bit MCF5482 ColdFire microprocessor contains a full/high-speed USB device controller and an Ethernet controller.

Windows Support

The modem driver included with Windows 98 SE and later (*usbser.sys*) is compatible with modems that use the Abstract Control Model. A modem used by applications that use the Windows Telephony Application Programming Interface (TAPI) must have its own INF file. Microsoft provides a Modem Development Kit with tools, sample INF files, and information for creating and testing INF files for modems.

Devices other than modems can use the *usbser.sys* driver as well. To enable host applications to access a device using COM-port functions (a virtual COM port), bInterfaceSubClass must be set to the Abstract Control Model. For better performance, however, most device developers use a driver from another source.

Under Windows 2000 and later, the *usb8023.sys* driver maps Remote NDIS to USB.

Content Security

The Content Security class defines a way for content owners to control access to files, music, video, or other data transmitted on the bus. The control can use either of two defined Content Security Methods: Basic Authorization or Digital Transmission Content Protection (DTCP).

Documentation

In addition to the main Content Security specification, each content security method (CSM) has its own specification document. At this writing, the latest edition of the specifications is 1.0. The DTCP specification and license information are available from the Digital Transmission Licensing Administrator (*www.dtcp.com*).

Overview

The class defines a protocol for activating and deactivating a content security method and for associating a content security method to a channel. A channel represents a relationship between an interface or endpoint and one or more CSMs. Only one CSM can be active on a channel at a time.

Basic Authorization, also known as Content Security Method 1, or CSM-1, consists only of the class-specific request Get_Unique_ID, which enables a host to request an ID value from a device.

CSM-2 is DTCP, which was developed to prevent unauthorized copying of audio and video entertainment content via USB and other buses. A content owner can use DTCP to specify whether copying is allowed, identify authorized users, and specify an encryption method. A DTCP interface must have an interrupt endpoint in each direction for sending and receiving event notifications. A content provider who wants to use DTCP must sign a license agreement and pay an annual (not trivial) fee.

Two additional CSMs that don't have USB specifications at this writing are Open Copy Protection System (CSM-3) and Elliptic Curve Content Protection Protocol (CSM-4).

Descriptors

A Content Security function is defined at the interface level, with bInterfaceClass = 0Dh to indicate the Content Security class.

There are four class-specific descriptors:

Descriptor Name	Description	Use
CS_GENERAL	Identifies the Content Security Interface version number.	One per interface
Channel	Identifies one or more CSMs for a channel, which can be specified by interface number endpoint address.	One per channel
Content Security Method	Describes a CSM implemented on a device.	One per CSM
Content Security Method Variant	Describes a variant of the associated CSM.	Not used by CSM-1 or CSM-2

CSM-2 also defines a String descriptor for the string "Digital Transmission Content Protection Version 1.00".

Class-specific Requests

Two class-specific requests apply to all CSM interfaces. Get_Channel_Settings enables the host to learn what CSM is assigned to a channel. The Set_Channel_Settings request enables the host to assign a CSM to a channel or deactivate a previously assigned CSM.

CSM-2 has additional control requests to transfer Authentication and Key Exchange (AKE) commands and responses.

Chips

For a device using content security, the choice of a USB controller depends mainly on the capabilities needed to exchange the content being protected. Adding a Content-Security function requires only the occasional use of the control endpoint and for CSM-2, two interrupt endpoints.

Windows Support

Windows doesn't include a driver for the Content Security class, except for one function. Under Windows XP and later, if a device has a CSM-1 interface, an application can call the DeviceIoControl function with the dwIoControlCode parameter set to this value:

IOCTL_STORAGE_GET_MEDIA_SERIAL_NUMBER

The function requests the device's serial number from Windows' generic parent driver.

Device Firmware Upgrade

The Device Firmware Upgrade (DFU) class defines a protocol to enable a host to send firmware enhancements and patches to a device. After receiving the firmware upgrade, the device re-enumerates using its new firmware.

Documentation

The *Device Firmware Upgrade* specification defines the class. At this writing, the current version is 1.0.

Overview

To perform a firmware upgrade as described in the specification, a device must have two complete sets of descriptors: run-time and DFU-mode. The run-time descriptors are for normal operation and also include descriptors that inform the host that the device is capable of firmware upgrades. The DFU-mode descriptors are a separate set of descriptors for use when the device is upgrading its firmware. For example, a keyboard using its run-time descriptors enumerates as a HID-class device and sends keypress data to the host. During a firmware upgrade, the device suspends normal operations as a keyboard and uses the DFU-mode descriptors to communicate with the DFU driver on the host.

The upgrade process has four phases. In the first phase, device enumeration, the device sends its run-time descriptors to the host and operates normally. In the reconfiguration phase, the host sends a DFU_Upgrade request and then resets and re-enumerates the device, which returns its DFU-mode descriptors. In the transfer phase, the host transfers the firmware upgrade to the device. The manifestation phase begins when the device informs the host that the upgrade has been received. The host resets the bus, and the device enumerates using its upgraded firmware and resumes normal operation. During the upgrade process, the device transitions through defined states such as dfuIdle (waiting for DFU requests) or dfuError (an error has occurred).

An upgrade file stored on the host contains the firmware for the upgrade, followed by a DFU suffix that the host can use to help ensure that the firmware is valid and appropriate for a particular device. The suffix contains an error-checking value, a signature consisting of the ASCII codes for the text “DFU”, and optional values for the Vendor ID, Product ID, and product release number the firmware is appropriate for. The suffix is for the host’s use only; the host doesn’t send the suffix to the device.

To ensure that the host will load a new driver for the firmware-upgrade process, the device should use different Product IDs in its run-time and DFU-mode device descriptors.

DFU communications use only the control endpoint.

Descriptors

The DFU function is defined at the interface subclass level. In a device that supports DFU, both the run-time and DFU-mode descriptors include a standard interface descriptor with bInterfaceClass = FEh to indicate an Application Specific class and bInterfaceSubClass = 01h to indicate the

Device Firmware Upgrade class. In DFU mode, the DFU interface must be the only active interface in the device.

Both descriptor sets include a Run-time DFU Functional descriptor that specifies whether the device can communicate on the bus immediately after the manifestation phase, how long to wait for a reset after receiving a DFU_Upgrade request, and the maximum number of bytes the device can accept in a control Write transfer during a firmware upgrade.

Class-specific Requests

There are seven class-specific requests:

Request	Description
DFU_Detach	If a bus reset occurs within the time period specified in the DFU Functional descriptor, enumerate using the DFU-mode descriptors.
DFU_Dnload	Accept new firmware in the request's Data stage. A request with wLength = 0 means that all of the firmware has been transferred.
DFU_Upload	Send firmware to the host in the request's Data stage.
DFU_GetStatus	Return status and error information. On error, enter the dfuError state.
DFU_ClrStatus	Clear the dfuError state reported in response to a DFU_GetStatus request and enter the dfuIdle state.
DFU_GetState	Same as DFU_GetStatus but with no change in state on error.
DFU_Abort	Return to the dfuIdle state.

Chips

The choice of USB controller depends mainly on the requirements of the device in run-time mode. The device must have enough memory and other resources to store and implement the upgraded firmware.

Windows Support

Windows doesn't provide a driver for this class. STMicroelectronics has a Windows driver and firmware examples for use with its ST7 microcontrollers with Flash memory.

Human Interface

The Human Interface Device (HID) class includes keyboards, pointing devices, and game controllers. With these devices, the host reads and acts on human input such as keypresses and mouse movements. Hosts must respond quickly enough so users don't notice a delay between an action and the expected response. Some devices that perform vendor-specific functions can also use the HID class.

All HID data travels in reports, which are structures with defined formats. Usage tags in a report tell the host or device how to use received data. For example, a Usage Page value of 09h indicates a button, and a Usage ID value tells which button, if any, was pressed.

Windows and other operating systems have included HID drivers beginning with the earliest editions with USB support. The availability of class drivers has helped to make the HID class popular for devices besides obvious human-interface applications. A HID can exchange any type of data, but can use only control and interrupt transfers. Chapter 11, Chapter 12, and Chapter 13 have more about using HIDs in custom devices.

Documentation

The HID specification is in several documents. At this writing, the current version of the HID specification is 1.11. The main change from version 1.0 is enabling the host to send reports in interrupt OUT transfers. In a HID 1.0 interface, the host must send all reports in control transfers.

Several documents define Usage-tag values for different device types. *HID Usage Tables* has values for keyboards, pointing devices, various game controllers, displays, telephone controls, and more. Four other device types have their own documents:

Class Definition for Physical Interface Devices (PID) defines values for force-feedback joysticks and other devices that require physical feedback in response to inputs.

The *Monitor Control* class specification defines values for user controls and power management for display monitors. (The HID interface controls the display's settings only. The image data uses a different hardware interface.)

Usage Tables for HID Power Devices defines values for Uninterruptible Power Supply (UPS) devices and other devices where the host monitors and controls batteries or other power components.

Point of Sale (POS) Usage Tables defines values for bar-code readers, weighing devices, and magnetic-stripe readers.

Overview

HIDs communicate by exchanging reports using control and interrupt transfers. Input and Output reports may use control or interrupt transfers. Feature reports use control transfers. A report descriptor defines the size of each report and Usage values for the report data.

Descriptors

A HID function is defined at the interface level. In the interface descriptor, bInterfaceClass = 03h to indicate the HID class. The bInterfaceSubClass field indicates whether the HID supports a boot protocol, which is a protocol that a host can use instead of the report protocol defined in the device's report descriptor. Mice and keyboards may support a boot protocol to enable using the devices before the full HID drivers are loaded.

Following the interface descriptor is a class-specific HID descriptor, which contains the size of the report descriptor. The report descriptor contains information about the data in the HID reports. An optional Physical Descriptor can describe the part(s) of the human body that activate a control.

Class-specific Requests

HIDs have six class-specific control requests to enable sending and receiving reports, setting and reading the Idle rate (how often the device sends a report if the data is unchanged), and setting or reading the currently active protocol (boot or report). To obtain a report descriptor or physical descriptor, the host sends a Get_Descriptor request to the interface with the high byte of wValue set to 01h to indicate a class-specific descriptor and the low byte of wValue set to 22h to request a report descriptor or 23h to request a physical descriptor.

Chips

For devices with a human interface, low speed is fast enough to enable acting on received user input with no detectable delay. Many HIDs use low speed because the device needs a more flexible and/or cheaper cable. A HID may use any speed, however.

A variety of controllers include additional support for keyboards, mice, and game controllers. Atmel Corporation's AT43USB325 contains an AVR microcontroller and a 5-port hub. One of the hub's ports connects to an embedded function with support for a 20 x 8 keyboard matrix. The controller supports low and full speeds. The AT43USB325 is similar but supports an 18 x 8 keyboard matrix. Other vendors with controllers designed for use in keyboards include Alcor Micro and Winbond Electronics Corporation. Some general-purpose controllers, such as Cypress' CY7C63743, support both USB and PS/2 interfaces to make it easy to design a dual-interface device.

Code Mercenaries offers programmed chips for use in pointing devices, keyboards, and joysticks. The MouseWarrior series has interfaces for sensors and buttons and supports four interfaces: USB, PS/2, asynchronous serial, and Apple Desktop Bus (ADB). The KeyWarrior series supports USB, PS/2, and ADB and has interfaces to keyboard matrixes and optional support for keyboard macros. The JoyWarrior series supports a variety of game-controller inputs.

Windows Support

Applications can communicate with HIDs using API functions. The API functions for exchanging reports include ReadFile and WriteFile as well as HID-specific APIs such HidD_SetFeature and HidD_GetFeature. Applications that access game controllers can use DirectX's DirectInput component for fast, more direct access.

Windows requests exclusive access to Input reports from system keyboards and pointing devices, so applications can't directly read the reports that describe keypresses, mouse movements, and mouse-button clicks. Instead, the operating system handles this data at a lower level. For example, a Visual-Basic application doesn't have to read mouse clicks to find out if a user has clicked on an option button because the button's click event executes automatically on a button click.

If a system has multiple keyboards or pointing devices, Windows treats them all as a single "virtual" keyboard or pointing device. If you want to limit the applications that can access a keyboard or pointing device, or if you want to determine which keyboard or pointing device is the source of input, you need to either provide a digitally signed filter driver or design a vendor-specific device that the host doesn't identify as a system mouse or keyboard.

IrDA Bridge

The IrDA (Infrared Data Association) interface defines hardware requirements and protocols for exchanging data over short distances via infrared energy. A USB IrDA bridge converts between USB and IrDA data and enables a host to use USB to monitor, control, and exchange data over an IrDA interface.

Documentation

The specification for USB IrDA bridges is *IrDA Bridge Device Definition*. The current version at this writing is 1.0. The IrDA specifications are available from *www.irda.org*.

Overview

The data in an IrDA link uses the Infrared Link Access Protocol (IrLAP), which defines the format of the IrDA frames that carry data, addresses, and status and control information. The IrLAP Payload consists of the address, control, and optional information (data) fields in an IrLAP frame. In addition to the IrLAP Payload, each frame contains an error-checking value and markers for the beginning and end of the frame.

A USB IrDA bridge uses bulk pipes to exchange data with the host. The host and bridge place status and control information in headers whose format is defined in the IrDA bridge specification On receiving data from the IrDA link, the IrDA bridge extracts the IrLAP Payload, adds a header, and passes the data and header to the host. The header can contain values for the IrDA link's Media_Busy and Link_Speed parameters. On receiving IrDA data from the host, the IrDA bridge removes the header added by the host. The header can specify new values for Link_Speed and the number of beginning-of-frame markers. The bridge then places the IrDA Payload in an IrDA frame for transmitting.

Descriptors

An IrDA-bridge function is defined at the interface subclass level. In the interface descriptor, bInterfaceClass = FEh to indicate an application-specific interface and bInterfaceSubclass 02h to indicate an IrDA Bridge Device. A class-specific descriptor contains IrDA-specific information such as the maximum number of bytes in an IrDA frame and supported Baud rates.

Class-specific Requests

There are five class-specific control requests:

Request	bRequest	Description
Receiving	1	Is the device currently receiving an IrLAP frame?
Check_Media_Busy	3	Is infrared traffic present?
Set_IrDA_Rate_Sniff	4	Accept frames at any speed or at a single speed.
Set_IrDA_Unicast_List	5	Accept frames from the named addresses only.
Get_Class_Specific_Descriptors	6	Return the class-specific descriptor.

Chips

SigmaTel, Inc.'s STIR4000 is an IrDA USB bridge chip that contains a full-speed USB transceiver and an interface to an IrDA transceiver. The host communicates with the chip by accessing a series of registers that enable configuring, obtaining status information, and exchanging data. The chip supports vendor-specific control requests for reading and writing to the registers. The STIR4200 is a high-speed version of the chip.

Another approach to adding IrDA to a USB host is to use a USB/asynchronous-serial converter with an IrDA interface. Texas Instruments' TUSB3410 is a USB/asynchronous-serial converter for use in wired and IrDA serial interfaces. For a wired link, the chip's internal UART interfaces to serial-data pins. For an IrDA link, the UART interfaces to an internal IrDA encoder/decoder, which in turn connects to an external IrDA transceiver.

Windows Support

Windows XP supports IrDA communications via two software profiles. The dial-up networking profile enables using IrDA to connect a PC and a mobile phone. The LAN access profile enables using the Point-to-Point Protocol (PPP), a direct peer-to-peer network connection, or a direct connection to a network access point. Windows XP doesn't include a generic driver for the USB-IrDA-bridge function, but SigmaTel provides a driver for use with their chips.

Mass Storage

The mass-storage class is for devices that transfer files in one or both directions. Typical devices are floppy, hard, CD, DVD, and Flash-memory drives. Cameras can use the mass-storage class to enable accessing picture files in a camera's memory. In Windows computers, devices that use the mass-storage driver appear as drives in My Computer and the file system enables users to copy, move, and delete files in the devices.

Documentation

The USB specification for mass storage devices is in four documents: an overview (version 1.2), specifications for the bulk-only transport protocol (version 1.0) and the control/bulk/interrupt (CBI) transport protocol (version 1.1) and commands for the Universal Floppy Interface (UFI) (version 1.0).

Each media type has an industry-standard command-block set to enable controlling devices and reading status information. These are specifications that define command-block sets for device types supported by the mass-storage class:

ATAPI CD/DVD devices use the *ATA/ATAPI* specification from *www.t13.org* and the *MultiMedia Command (MMC) Set* from *www.t10.org*. (An earlier version of the ATA/ATAPI specification was called *SFF 8020i*.)

ATAPI removable media uses *SFF-8070i: ATAPI Removable Rewritable Media Devices*, available from *www.sffcommittee.com*. This document is a supplement to the ATA/ATAPI specification. Floppy drives often belong to this subclass.

Generic SCSI media uses the mandatory commands from the *SCSI Primary Command (SPC) Set* and *SCSI Block Command (SBC) Set* from *www.t10.org*.

QIC-157 tape drives use the *Common SCSI/ATAPI Command Set for Streaming Tape*, available from *www.qic.org*.

UFI uses the *UFI Command Specification* from *www.usb.org*. The commands are based on the SCSI-2 and SFF-8070i command sets.

Overview

Mass-storage devices use bulk transfers to exchange data. Control transfers send class-specific requests and can clear Stall conditions on bulk endpoints. For exchanging other information, a device may use either of two transport protocols: bulk only or control/bulk/interrupt (CBI). CBI is approved for use only with full-speed floppy drives. Bulk-only is recommended for new devices of all types.

In the bulk-only protocol, a successful data transfer has three stages: command transport, data transport, and status transport. In the command-transport stage, the host sends a command in a structure called a Command Block Wrapper (CBW). In the data-transport stage, the host or device sends the requested data. In the status-transport stage, the device

Table 7-7: The CBW contains a command block and other information about the command.

Name	Bits	Description
dCBWSignature	32	The value 43425355h, which identifies the structure as a CBW.
dCBWTag	32	A tag that associates this CBW with the CSW the device will send in response.
dCBWDataTransferLength	32	The number of bytes the host expects to transfer in the data-transport stage.
bmCBWFlags	8	Specifies the direction of the data-transport stage. Bit 7 = 0 for an OUT (host-to-device) transfer. Bit 7 = 1 for an IN (device-to-host) transfer. All other bits are zero. If there is no data-transport stage, bit 7 is ignored.
Reserved	4	0
bCBWLUN	4	For devices with multiple LUNs, specifies the LUN the command block is directed to. Otherwise the value is zero.
Reserved	3	0
bCBWCBLength	5	The length of the command block in bytes (1–16)
CBWCB	128	The command block for the device to execute.

sends status information in a structure called a Command Status Wrapper (CSW). Some commands have no data-transport stage.

Table 7-7 shows the fields in the CBW, which is 31 bytes. The meaning of the command-block value in the CBWCB field varies with the command set specified by the interface descriptor's bInterfaceSubClass field.

On receiving a CBW, a device must check that the structure is valid and has meaningful content. A CBW is valid if it is received after a CSW or reset, is 31 bytes, and has the correct value in dCBWSignature. The contents are considered meaningful if no reserved bits are set, bCBWLUN contains a supported LUN value, and bCBWCBLength and CBWCB are valid for the interface's subclass.

Table 7-8 shows the fields in the CSW, which is 13 bytes. On receiving a CSW, a device must check that the structure is valid and has meaningful content. A CSW is valid if it has 13 bytes, has the correct value in

Table 7-8: The CSW contains status and related information about a command.

Name	Bits	Description
dCBWSignature	32	The value 53425355h, which identifies the structure as a CSW.
dCBWTag	32	The value of the dCBWTag in a CBW received from the host.
dCSWDataResidue	32	For OUT transfers, the difference between dCBWDataTransferLength and the number of bytes the device processed. For IN transfers, the difference between dCBWDataTransferLength and the number of bytes the device sent.
bCSWStatus	8	00h = command passed 01h = command failed 02h = phase error

dCSWSignature, and has a dCSWTag value that matches dCBWTag of a corresponding CBW. The contents are considered meaningful if bCSWStatus equals 02h or if bCSWStatus equals either 00h or 01h and dCSWDataResidue is less than or equal to dCBWDataTransferLength.

Descriptors

The mass-storage function is defined at the interface level. In the device's interface descriptor, bInterfaceClass = 08h to indicate that the interface belongs to the mass-storage class.

The bInterfaceSubClass field indicates the supported command-block set:

bInterfaceSubClass	Subclass Description
02h	ATAPI CD/DVD devices
03h	QIC-157 tape devices
04h	USB Floppy Interface (UFI)
05h	ATAPI removable media
06h	Generic SCSI media

The bInterfaceProtocol field indicates the supported transport protocol:

bInterfaceProtocol	Protocol Description
00h	CBI with command completion interrupt transfers
01h	CBI without command completion interrupt transfers
50h	bulk only

Every bulk-only mass-storage device must have a serial number of at least 12 characters using only the characters in the range 0–9 and A–F. The serial number enables the operating system to retain properties such as the drive letter and access policies after a user moves a device to another port or attaches multiple devices with the same Vendor ID and Product ID. The device descriptor's iSerialNumber field contains an index to the serial number, which is stored in a string descriptor. The value must be different from any serial number used by other devices with the same values in the idVendor, idProduct, and bcdDevice fields in the device descriptor.

A mass-storage device must have a bulk endpoint for each direction.

Class-specific Requests

The bulk-only protocol has two defined control requests: Bulk Only Mass Storage Reset (reset the device) and Get Max Lun (get the number of logical units, or partitions, that the device supports). All other commands and status information travel in bulk transfers.

The control/bulk/interrupt (CBI) protocol has one defined control request: Accept Device-Specific Command (ADSC). The Data stage of the request carries the command. A device can use an interrupt transfer to indicate that the device has completed a command's requested action.

Chips

A mass-storage device can use just about any full- or high-speed controller chip, but several manufacturers have controllers designed specifically for use in mass-storage devices. Prolific Technology and Standard Microsystems Corporation (SMSC) each have a variety of chips with interfaces to a variety of mass-storage device types. Controllers with direct interfaces to

ATA/ATAPI devices include Philips Semiconductor's ISP1183, Texas Instruments' TUSB6250, and Cypress Semiconductor's EZUSB AT2.

Windows Support

Windows 2000 and later include a driver that supports bulk-only and CBI devices. When a device's descriptors identify the device as mass-storage class, the operating system loads the USB storage port driver (*usbstor.sys*). This driver manages communications between the lower-level USB drivers and Windows' storage-class drivers. When the device is formatted using a supported file system, the operating system assigns a drive letter to the device and the device appears in My Computer.

The mass-storage driver in Windows XP supports bInterfaceSubClass codes 02h, 05h, and 06h. Support for drives with multiple Logical Unit Numbers (LUNs) was added in Windows 2000 SP3.

One point of confusion relating to the mass-storage support under Windows is the difference between removable devices and removable media. All USB drives are removable devices because they're easily attached and detached from the PC. A removable device may have removable or non-removable media. CD, DVD, and floppy drives have removable media. A hard disk is a non-removable medium because you can't easily remove the disk from the drive. Windows' Autorun capability (also called AutoPlay) applies to devices with removable media. Autorun enables the operating system to run a program, play a movie, or perform other actions when a disk or other removable media is inserted.

Printers

The printer class is for devices that convert received data into text and/or images on paper or other media. The most basic printers print lines of text in a single font. Most laser and inkjet printers understand one or more page description languages (PDLs) and can print text in any font and complex images.

Documentation

The USB Printing Devices specification is for printers of all types. At this writing, the current version of the specification is 1.1. The IEEE-1284 standard from *www.ieee.org* describes the interface used by parallel-port printers and includes information, such as the format for Device IDs, used by USB printers.

Overview

Printer data uses a bulk OUT pipe. The host obtains status information in control requests or an optional bulk IN pipe.

Descriptors

The printer function is defined at the interface level. In the interface descriptor, bInterfaceClass = 07h to specify the printer class.

The interface descriptor's bInterfaceProtocol field contains a value that names a type of printer interface:

bInterfaceProtocol	Type
01h	Unidirectional
02h	Bidirectional
03h	IEEE-1284.4-compatible Bidirectional

With all three interface protocols, the host uses the bulk OUT endpoint to send data to the printer. With the unidirectional protocol, the host retrieves status information by sending a class-specific Get_Port_Status request. With the bidirectional protocol, the host can retrieve status information using Get_Port_Status or the bulk IN pipe, which can provide more detailed information. The IEEE-1284.4-compatible bidirectional protocol is like the bidirectional protocol but with added support to enable communications with individual functions in a multifunction peripheral.

Class-specific Requests

The printer class has three class-specific requests:

Request	bRequest
Get_Device_ID	0
Get_Port_Status	1
Soft_Reset	2

In response to a GET_DEVICE_ID request, the device returns a Device ID in the format specified by the IEEE-1284 standard. The first two bytes of the Device ID are the length in bytes, most significant byte first. Following the length is a string containing a series of keys and their values in this format:

```
key: value {,value};
```

All Device IDs must contain the keys MANUFACTURER, COMMAND SET, and MODEL, or their abbreviated forms (MFG, CMD, and MDL). The COMMAND SET key names any PDLs the printer supports, such as Hewlett Packard's Printer Control Language (PCL) or Adobe Postscript. Additional keys, which may be vendor-defined, are optional.

Here is an example Device ID:

```
MFG:My Printer Company;
MDL:Model 5T;
CMD:MLC,PCL,PML;
DESCRIPTION:My Printer Company Laser Printer 5T;
CLASS:PRINTER;
REV:1.3.2;
```

In response to the GET_PORT_STATUS request, the device returns a byte that emulates the Status-port byte on a parallel printer port. Three bits in the byte contain status information:

Bit	Name	meaning when = 1	meaning when = 0
3	Not Error	no error	error
4	Select	printer selected	printer not selected
5	Paper Empty	out of paper	not out of paper

A printer that can't obtain the status information should respond with 18h to signify *no error, printer selected*, and *not out of paper*. Parallel-port printers have two additional status bits, Busy and Ack, which are used in handshaking and don't apply to USB printers.

On receiving a Soft_Reset request, a device should flush all buffers, reset the interface's bulk pipes to their default states, and clear all Stall conditions.

In a Soft_Reset request, the bmRequestType value in the Setup transaction should be 21h to signify a class-specific request that is directed to an interface and has no Data stage. However, version 1.0 of the printer-class specification incorrectly listed the bmRequestType for Soft_Reset as 23h. So to be on the safe side, devices should respond to hosts that use a bmRequestType of 23h with this request, and hosts should try the incorrect value on receiving a STALL in response to this request using the correct value.

Chips

Just about any full- or high-speed controller will have the one or two bulk endpoints for a printer function. For converting parallel-port printers to USB, Prolific Technology has the PL-2305 USB-to-IEEE-1284 Bridge Controller. The chip supports three endpoints: one bulk IN, one bulk OUT, and one interrupt IN. The chip's IEEE-1284 parallel port can interface to an existing parallel port on a printer or other peripheral.

Windows Support

Windows includes drivers that handle tasks common to both non-Postscript and Postscript printers. A printer manufacturer can customize a driver for a specific printer by providing a minidriver that consists of one or more text files with the customization information. The Windows DDK has information on how to create printer minidrivers.

When an application requests to print a file, the printer driver sends the printer data to the print spooler's print processor. If the printer has a USB interface, the print processor sends the data either directly to the *Usbmon* port driver or to a language monitor that modifies the data stream and passes it on to *Usbmon*. *Usbmon* in turn communicates with lower-level USB drivers that access the port.

Usbmon and the *Usbprint* driver provide a software interface that is similar to the interface for accessing parallel-port printers. In many cases, a printer can use the same printer driver and language monitor for both parallel-port and USB interfaces. If needed, a language monitor or other upper-level software can support USB-specific, vendor-specific requests.

Still Image Capture: Cameras and Scanners

The still-image class encompasses cameras that capture still images (in other words, not video) and scanners. The main job of a still-image device's USB interface is to transfer image data from the device to the host. Some devices can receive image data from the host as well. If all you need is a way to transfer image files from a camera, another option is to use the mass-storage driver.

Documentation

The USB class specification, *Still Image Capture Device Definition*, includes features and commands from *PIMA 15740: 2000 Picture Transfer Protocol*, which describes requirements for transferring files and controlling digital still cameras. At this writing, the current version of the still-image specification is 1.0. The PIMA document is available from the International Imaging Industry Association (I3A) at *www.i3a.org*.

Overview

A still-image device has one bulk IN endpoint and one bulk OUT endpoint for transferring both image data and non-image data. The specification also requires an interrupt IN endpoint for event data.

In the bulk and interrupt pipes, information travels in structures called containers. The four container types are the Command Block, Data Block, Response Block, and Event Block. The bulk OUT pipe carries Command and Data Blocks. The bulk IN pipe carries Data and Response Blocks. The interrupt IN pipe carries Event Blocks.

On the bulk pipes, the host communicates by using a protocol with three phases: Command, Data, and Response. A short packet indicates the end of a phase. In the Command phase, the host sends a Command Block that names an operation defined in PIMA 15740. The Command Block contains an operation code that determines if the operation requires a data transfer and if so, the direction of data transfer. If there is a data transfer, the data travels in a Data Block in the Data phase. The first four bytes of the Data Block are the length in bytes of the data being transferred. Some operations have no Data phase. The final phase is the Response phase, where the device sends a Response Block containing completion information.

On the interrupt pipe, an Event Block can contain up to three Event Codes with status information such as a low-battery warning or a notification that a memory card has been removed. The Check Device Condition Event Code requests the host to send a class-specific Get_Extended_Event_Data request for more information about an event.

A device using the bulk-only protocol cancels a transfer by stalling the bulk endpoints. The host then sends a class-specific Get_Device_Status request and uses the Clear_Feature request to clear the stalled endpoints. The host cancels a transfer by sending a class-specific Cancel_Request request. A device is ready to resume data transfers when it returns OK (PIMA 15740 Response Code 2001h) in response to a Get_Device_Status request.

Descriptors

A still-image function is defined at the interface level. In the interface descriptor, bInterfaceClass = 06h to indicate a still-image device, bInterfaceSubclass = 01h to indicate an image interface, and bInterfaceProtocol = 01h to indicate a still-image capture function. The interface must have descriptors for the bulk IN, bulk OUT, and interrupt IN endpoints.

Class-specific Requests

There are four class-specific control requests:

Request	bRequest	Required?
Cancel_Request	64h	yes
Get_Extended_Event_Data	65h	no
Device_Reset_Request	66h	yes
Get_Device_Status	67h	no

With Cancel_Request, the host requests to cancel the PIMA 15740 transaction named in the request. With Get_Extended_Event_Data, the host requests extended information regarding an event or vendor condition. With Device_Reset_Request, the host requests the device to return to the Idle state. The host can use this request after a bulk endpoint has returned a STALL or to clear a vendor-specific condition. With Get_Device_Status, the host requests information needed to clear halted endpoints. The host uses this request after a device has canceled a data transfer.

Chips

Just about any full- or high-speed USB controller will have the three endpoints required by the still-image class.

Windows Support

Recent Windows editions support the Windows Image Acquisition (WIA) API for communicating with devices in the still-image class. Applications communicate with devices by using ReadFile, WriteFile, and DeviceIoControl commands. The drivers that add USB support to WIA are *usbscan.sys* in Windows XP and later and *usbscn9x.sys* in Windows Me.

Under Windows XP, cameras that use the Picture Transfer Protocol (PTP) described in the PIMA 15740 standard require no vendor-provided driver components, though vendors can provide a minidriver to enhance the driver and support vendor-specific features and capabilities. For scanners, the vendor must provide a microdriver, which is a “helper DLL” that translates between the driver’s communications and a language the scanner understands, or a minidriver to work with the provided drivers to enable communications with the device.

Windows 98 and Windows 2000 use an earlier Still Image architecture (STI). Product vendors must provide a user-mode driver to work with the provided STI driver.

Test and Measurement

The test-and-measurement class (USBTMC) is suited for instrumentation devices where the data doesn't need guaranteed timing. These devices typically contain components such as ADCs, DACs, sensors, and transducers. A device may be a stand-alone unit or a card in a larger computer.

Before USB, many test-and-measurement devices used the IEEE-488 parallel interface, also known as the General Purpose Interface Bus (GPIB). The USB488 subclass of the test-and-measurement class defines protocols for communicating using IEEE-488's data format and commands.

Documentation

The class's specifications include the main *Test and Measurement Class* specification and a separate document for the USB488 subclass. At this writing, the current version of both documents is 1.0. The IEEE-488 standards are available from *www.ieee.org*.

Overview

A test-and-measurement device requires a bulk OUT endpoint and a bulk IN endpoint. An interrupt IN endpoint is required for devices in the USB488 subclass and otherwise is optional for returning event and status information.

The bulk pipes exchange messages, with each message consisting of a header followed by data. The bulk OUT endpoint receives command messages, and the bulk IN endpoint sends response messages. The header for a command message contains a message ID, a bTag value that identifies the transfer, and message-specific information. The header for a response message contains the message ID and bTag values of the command that prompted the response, followed by message-specific information. The message ID specifies whether a command is device-dependent or vendor-specific and whether the host expects a response.

Descriptors

A test-and-measurement function is specified at the interface subclass level. In the interface descriptor, bInterfaceClass = FEh to indicate an application-specific interface and bInterfaceSubClass = 03h to indicate the test-and-measurement class. There are no class-specific descriptors.

Class-specific Requests

The class defines eight control requests for controlling and requesting the status of an interface or transfer and requesting information about the interface's attributes and capabilities.

Chips

Just about any full- or high-speed device will have the two or three endpoints this class requires.

Windows Support

Windows doesn't include a driver for this class. National Instruments provides a driver for use with its hardware. Other options for test-and-measurement devices that use bulk transfers include the mass-storage class or a vendor-specific driver. A HID-class device can also perform test and measurement functions. For an existing device with an IEEE-488 interface, the quick solution is to use a commercial IEEE-488/USB converter.

Video

The video class supports digital camcorders, webcams, and other devices that send, receive, or manipulate transient or moving images. The class also supports transferring still images from video devices. Because transmitting high-quality video requires a lot of bandwidth, using USB for video has become a more attractive option since high-speed hosts and devices have become available.

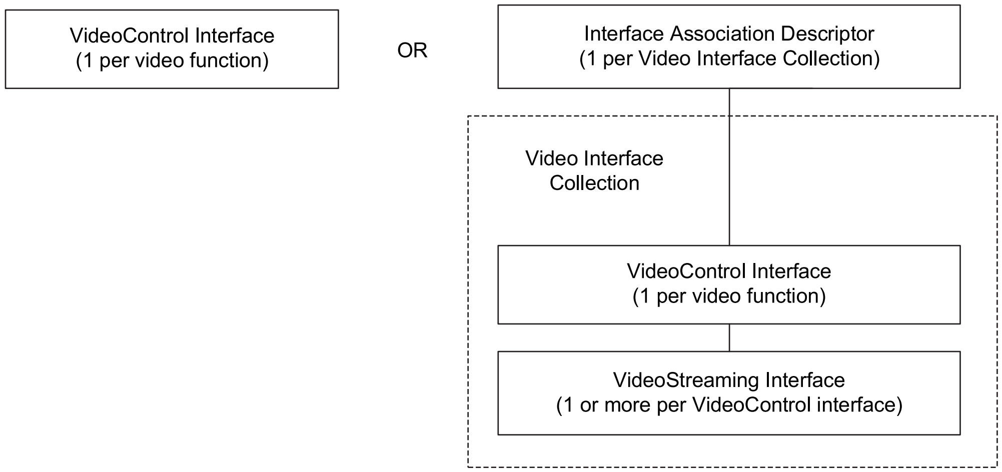
VideoControl Interface
(1 per video function)
OR
Interface Association Descriptor
(1 per Video Interface Collection)
Video Interface Collection
VideoControl Interface
(1 per video function)
VideoStreaming Interface
(1 or more per VideoControl interface)

Figure 7-6: A video interface consists of a VideoControl interface and zero or more VideoStreaming interfaces.

Documentation

A variety of documents make up the video specification. The *Video Class Definition* defines standard and class-specific descriptors and class-specific control requests for video devices. The *Media Transport Terminal* specification defines descriptors and requests for devices such as video cameras and digital VCRs, which stream data stored in sequential media and may require functions such as play, record, rewind, and eject. Separate payload specifications contain format-specific information for a variety of video formats such as MJPEG, MPEG2-TS, DV, and uncompressed video. Version 1.1 of the video class specification (under development at this writing) will retire some additional 1.0 formats and add generic frame-based and generic stream-based formats. Other specification documents include a video camera example, an FAQ, and an Identifiers document that gathers together identifier values defined in the other video-class specifications. At this writing, the current version of all of these specifications is 1.0.

Overview

Figure 7-6 shows the elements that make up a video function in a USB device. Every function must have a VideoControl interface, which provides information about inputs, outputs, and other components of the function. Most functions also have one or more VideoStreaming interfaces that enable transferring video data. A Video Interface Collection consists of a VideoControl interface and its associated VideoStreaming interfaces. (A function with only a VideoControl interface isn't part of a Video Interface Collection.) A device can have multiple, independent VideoControl interfaces and Video Interface Collections.

The VideoControl interface uses the control endpoint and may use an interrupt IN endpoint. Each VideoStreaming interface has one isochronous or bulk endpoint for video data and an optional bulk endpoint for still-image data.

Descriptors

The video class defines an extensive set of descriptors that enable devices to provide detailed information about the device's abilities. Each Video Interface Collection must have an interface association descriptor that specifies the interface number of the first VideoControl interface and the number of VideoStreaming interfaces associated with the function.

The VideoControl Interface. The VideoControl interface (Figure 7-7) has a standard interface descriptor with bInterfaceClass = 0Eh to indicate the video class, plus a class-specific VideoControl interface descriptor, which consists of a VideoControl interface header descriptor followed by one or more Terminal and/or Unit descriptors. A Terminal is the starting or ending point for information that flows into or out of a function. A Terminal may represent a USB endpoint or another component such as a CCD sensor, display module, or composite-video input or output. A Terminal descriptor can describe an Input Terminal or Output Terminal. The descriptor's wTerminalType field names the function of the terminal the descriptor is associated with, such as camera, media transport input, or media transport output. A Unit transforms data flowing through a function. There are three types of Unit descriptors: Selector Unit for routing a data stream to an output, Processing Unit for controlling video attributes, and Extension Unit for vendor-defined functions.

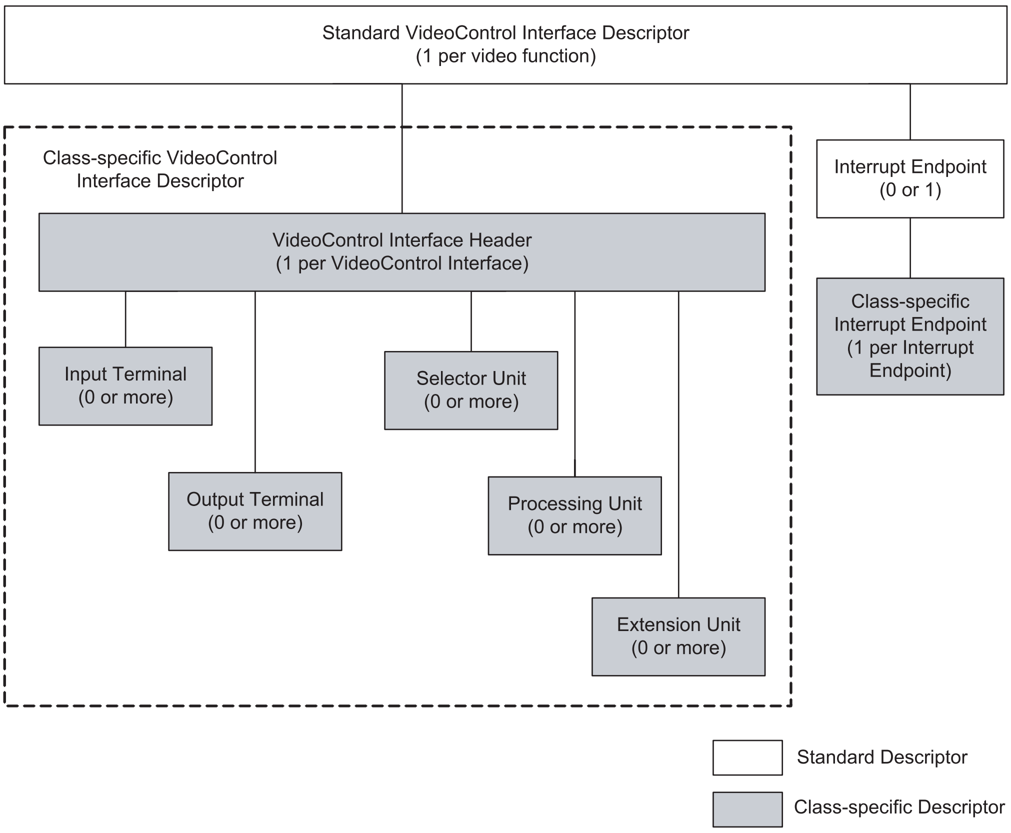
Standard VideoControl Interface Descriptor
(1 per video function)
Class-specific VideoControl Interface Descriptor
VideoControl Interface Header
(1 per VideoControl Interface)
Input Terminal
(0 or more)
Output Terminal
(0 or more)
Selector Unit
(0 or more)
Processing Unit
(0 or more)
Extension Unit
(0 or more)
Interrupt Endpoint
(0 or 1)
Class-specific Interrupt Endpoint
(1 per Interrupt Endpoint)
Standard Descriptor
Class-specific Descriptor

Figure 7-7: The VideoControl interface provides information about inputs, outputs, and other components of a video function.

If the interface has an interrupt endpoint, the endpoint has a standard endpoint descriptor followed by a class-specific endpoint descriptor.

The VideoStreaming Interface. Each VideoStreaming interface (Figure 7-8) has a standard interface descriptor. Following the standard interface descriptor, an interface with an IN endpoint has a class-specific VideoStreaming Input Header descriptor, and an interface with an OUT endpoint has a class-specific VideoStreaming Output Header descriptor.

Following the Header descriptor is a Payload Format descriptor for each supported video format. For frame-based formats, the Payload Format descriptor is followed by one or more Frame descriptors that describe the

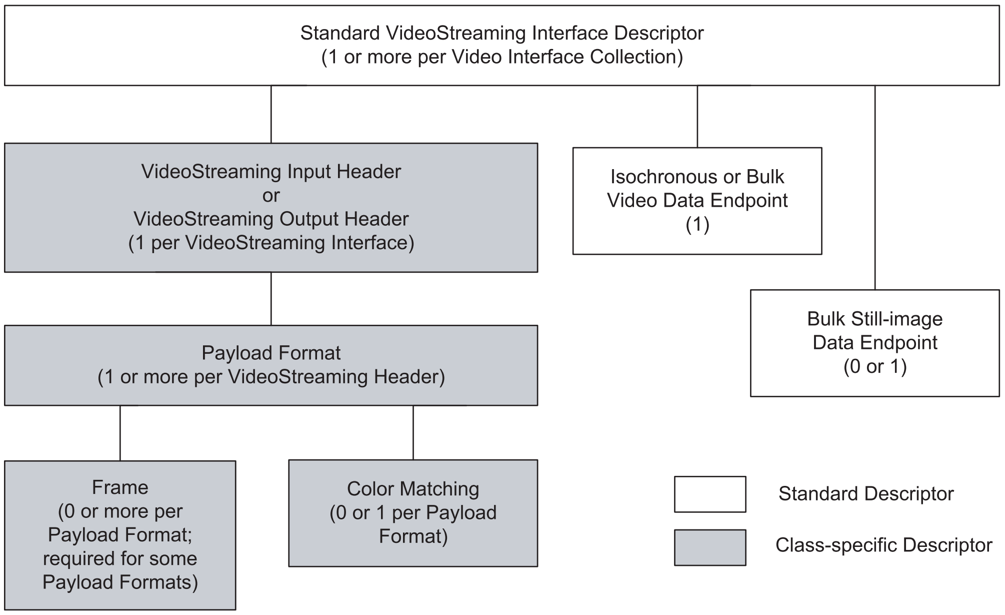
Standard VideoStreaming Interface Descriptor
(1 or more per Video Interface Collection)
VideoStreaming Input Header
or
VideoStreaming Output Header
(1 per VideoStreaming Interface)
Isochronous or Bulk
Video Data Endpoint
(1)
Bulk Still-image
Data Endpoint
(0 or 1)
Payload Format
(1 or more per VideoStreaming Header)
Frame
(0 or more per
Payload Format;
required for some
Payload Formats)
Color Matching
(0 or 1 per Payload
Format)
Standard Descriptor
Class-specific Descriptor

Figure 7-8: A VideoStreaming interface has an endpoint for video data and an optional endpoint for still-image data.

dimensions of the video frames and other characteristics specific to a format. A Payload Format can also have a Color Matching descriptor to describe a color profile. Each VideoStreaming interface has one isochronous or bulk endpoint descriptor for video data and an optional bulk endpoint descriptor for still-image data.

Class-specific Requests

Class-specific control requests enable setting and reading the states of controls in VideoControl and VideoStreaming interfaces.

Chips

Vista Imaging's ViCAM-III chip contains a programmable digital imaging engine with extensive support for video functions and a USB controller. Cypress Semiconductor has partnered with several companies to offer reference designs that use EZ-USB controllers in various video applications.

Windows Support

A driver compatible with the video class (*usbvideo.sys*) was released in Windows XP SP2. Vendors of video-class devices that use the driver don't need to provide any driver software but can provide a Control or Streaming extension to support vendor-specific functions or features.

Applications can access video devices using the DirectShow component of DirectX. The *usbvideo.sys* driver is supported beginning with DirectX version 9.2.

For earlier Windows editions, manufacturers of video devices must provide a minidriver to specify a format for streaming video, implement device-specific functions and properties, and perform bulk transfers if required for video data. Windows' USBCAMD driver manages isochronous data transfers, including synchronizing, starting, and stopping communications and recovering from errors. The driver communicates with Windows' stream-class driver and with the lower-level USB drivers.

Implementing Non-standard Functions

Some devices don't have an obvious match to a defined class. Examples include some data-acquisition devices and controllers for motors, relays, or other circuits. Another common application that doesn't fit into an obvious class is linking two hosts. Before USB, these types of applications used the legacy serial and parallel ports. USB is flexible enough to accommodate these and other vendor-specific applications.

Standard or Custom Driver?

When possible, it's almost always preferable to use a class that has drivers provided by the operating systems the device will operate under. Using a provided driver saves much time and effort.

Some devices with vendor-specific functions can be designed as HIDs. A HID doesn't have to be a standard device type and doesn't even need a human interface. The only requirements are that the descriptors must meet the class's requirements, and the device must transfer data using only interrupt or control transfers as defined in the HID specification.

The mass-storage class is another option for devices that exchange data in files and support a file system the host understands.

Some devices need to provide their own drivers. Using a driver provided by a chip manufacturer is one option. This approach saves you from having to develop a driver but leaves you dependent on the chip vendor to fix bugs and keep up with new operating-system editions. Chapter 8 has more about creating custom drivers.

Converting from RS-232

The RS-232 serial port has been with the PC since its beginning. The port has been used in thousands of peripherals. Just about any device that uses RS-232 can be implemented with USB. There are several approaches to making the switch.

First determine if the device fits into a defined class. Modems should use the communication-device class. Pointing devices, uninterruptible power supplies, and point-of-sale devices should be designed as HIDs.

For many other devices, FTDI Chip's FT232BM USB UART introduced in Chapter 6 provides a quick way to upgrade a design to USB. The chip can convert an existing RS-232 device to USB with minimal design changes and in most cases no changes to host software.

Figure 7-9 shows an example. A typical device with an RS-232 interface contains a UART that converts between the serial data used in RS-232 communications and the parallel data the CPU uses. The signals on the line side of the UART connect to converters that translate between RS-232 voltages and the 5V logic used by the UART. The line side of the converter connects to a cable to the remote computer with an RS-232 interface. To convert from RS-232 to USB, you replace the RS-232 converter with a '232BM. On the host computer, FTDI Chip's Virtual COM port driver enables applications to access the device using the same functions used for RS-232 communications.

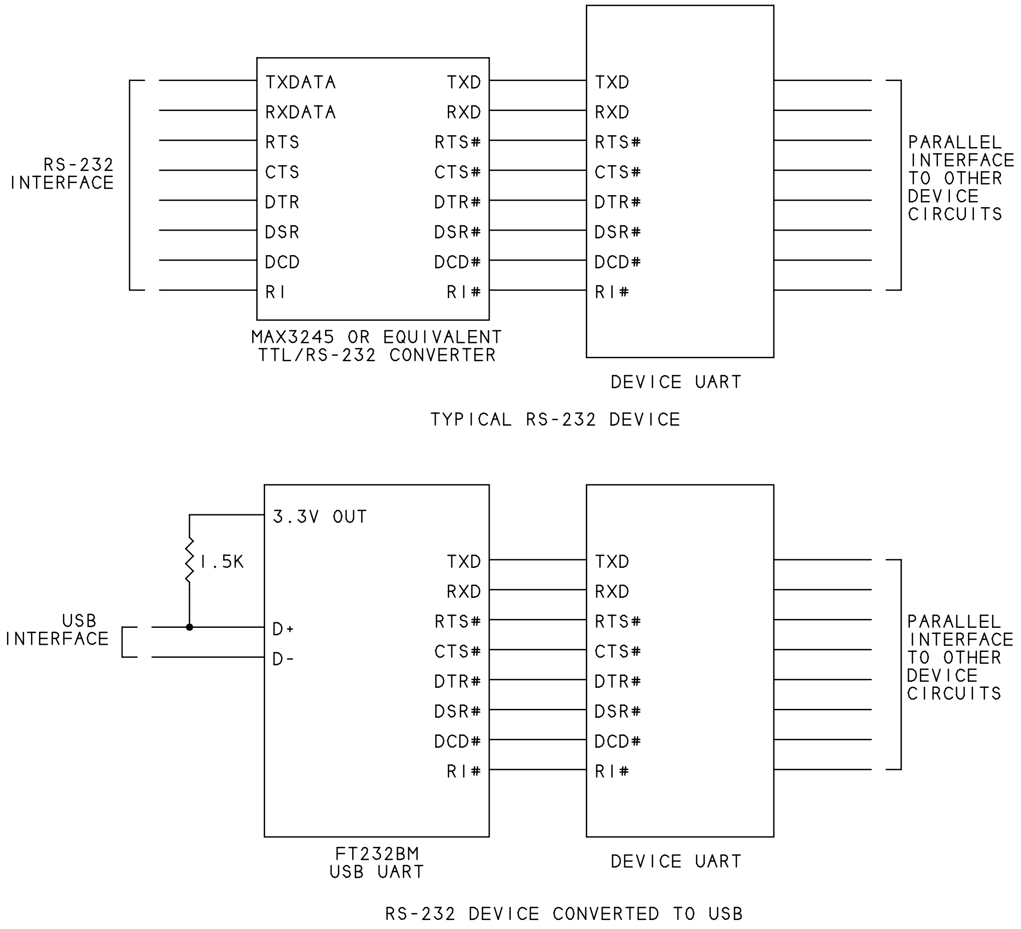

Figure 7-9: FTDI's USB UART can convert devices with RS-232 interfaces to USB. A driver provided by FTDI causes the device to appear like a conventional COM-port device to host applications.

An even easier solution is to use an RS-232/USB converter module. These typically contain little more than an FT232BM or similar chip, an RS-232 interface chip, an RS-232 connector, and a USB connector. Users then have a choice of using the original interface or adding the converter and using USB.

When using a USB/RS-232 converter, devices that use the status and control signals in unconventional ways and with critical timing requirements may require modifications to device hardware or firmware or application software.

Converting from the Parallel Port

Another port that all PCs had from the beginning was the parallel port. The port was originally intended for connecting printers, but many other device types took advantage of the port as well. The parallel interface is faster than RS-232 and thus became a favored connection for scanners and external drives. Scanners, drives, and printers can now use USB and the standard classes for these device types.

For other devices, there are several options for converting to USB. A peripheral-side parallel-port interface has 8 bidirectional data pins, 5 status outputs, and 4 control inputs. A USB controller with 17 or more I/O bits can emulate a parallel port. Prolific Technology's PL-2305 has a USB interface and a complete PC-side IEEE-1284 parallel port that can interface directly to existing parallel-port devices.

For the firmware and driver, devices that can function using only control and interrupt transfers may be able to use the HID class. The device will need new application software to communicate with the HID drivers in place of the driver that accessed the parallel port. If you want to make minimal changes to the application software, you can provide a custom driver that provides functions that emulate the functions called by the original application.

PC-to-PC Communications

Every USB communication must be between a host and a device. USB doesn't allow hosts to exchange data with each other directly. Yet because every PC has a USB port, it's natural to want to use the interface to connect PCs to each other, especially when the PCs don't have Ethernet ports.

USB On-The-Go enables a device to also function as a host. Most PCs don't contain On-The-Go host controllers, however. Another solution is to use a

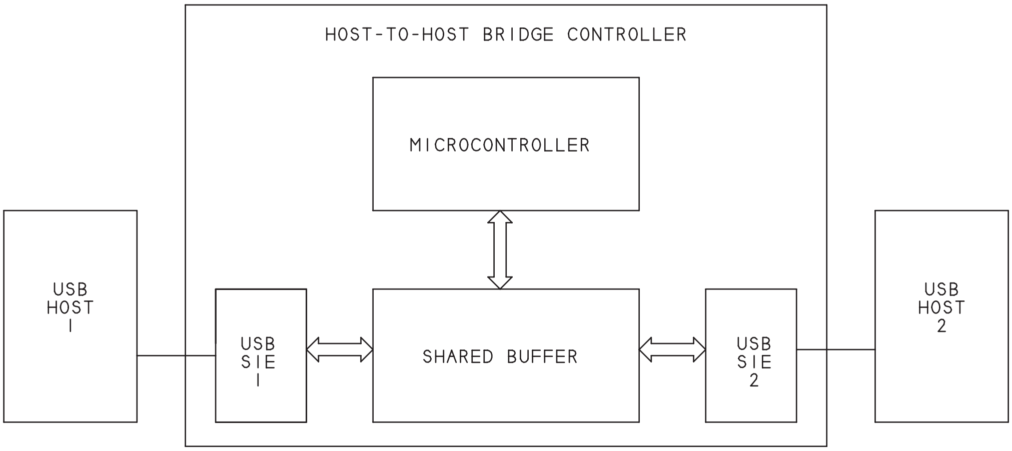

Figure 7-10: To enable two USB hosts to communicate with each other, two USB serial interface engines can share a buffer. Each SIE copies received USB data into the shared buffer, and the other device retrieves the data from the buffer and sends the data to the other host.

host-to-host bridge cable that contains two serial interface engines. Each PC has a USB connection to one of the SIEs, and the two devices communicate with each other via a shared buffer (Figure 7-10). Data sent by a PC travels to one of the SIEs, through the shared buffer, and out the other SIE to the remote PC.

Prolific Technology's PL-2501 Hi-Speed USB Host to Host Bridge Controller is a single chip designed for this type of host-to-host application. The chip contains an 8032 microcontroller and two USB SIEs that can access a common buffer. The PL-2301 is a full-speed version. Many commercial "data-link file-transfer cables" contain one of these chips. Typically, the drivers enable each PC to see the other as a network-connected computer.

An alternate approach is to use two FTDI Chip USB UARTs and cross-connect the asynchronous interfaces in a "null modem" configuration. The PCs then see each other as COM-port devices. Yet another option is to establish a network connection by attaching a USB/Ethernet converter to each PC and connecting each converter to a local network.

Using a Generic Driver

For devices that don't fit into a standard class, a generic driver can be a solution. Generic drivers typically enable applications to request control, interrupt, bulk, and isochronous transfers using a driver-specific API. Two such options are the DriverX USB toolkit from Tetradyne Software, Inc. and the USBIO Development Kit from Thesycon Systemsoftware & Consulting GmbH. (Yes, that spelling is correct.)

The DriverX USB toolkit includes a generic driver, header and library files for use with Visual C++ and Borland C++ Builder, and additional support for Delphi and Visual Basic.

To communicate with the driver included with the USBIO Development Kit, applications can use standard Windows API functions (ReadFile, WriteFile, DeviceIoControl), a C++ class library, native Delphi and Java interfaces, or a USBIO COM interface based on Microsoft's Component Object Model (COM) technology.