USB Device Interface Guide

Drivers, Kernel, & Hardware: User-Space Device Access

2007-09-04

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction **Introduction to USB Device Interface Guide 7**

Organization of This Document 7
See Also 7

Chapter 1 **USB Device Overview 9**

USB Device Types and Bus Speeds 9
USB Device Architecture and Terminology 10
- USB Device Component Descriptors 10
- USB Composite Class Devices 11
- USB Transfer Types 11
- Stalls and Halts 12
- Data Synchronization in Non-Isochronous Transfers 12
- USB 2.0 and Isochronous Transfers 12

USB Devices on Mac OS X 13
- Finding USB Devices and Interfaces 14
- USB Family Error Codes 15
- Determining Which Interface Version to Use 16

Tasks and Caveats 16
- Handling Stalls, Halts, and Data Toggle Resynchronization 16
- Using the Low Latency Isochronous Functions 17
- Errors Reported by the EHCI Hub 18
- Changes in Isochronous Functions to Support USB 2.0 18

USB Device Access in an Intel-Based Macintosh 19

Chapter 2 **Working With USB Device Interfaces 21**

Using USB Device Interfaces 21
Accessing a USB Device 22
- Definitions and Global Variables 23
- The main Function 23
- Working With the Raw Device 26
- Working With the Bulk Test Device 30
- Working With Interfaces 31

Document Revision History 39

Tables and Listings

Chapter 1 **USB Device Overview 9**

Table 1-1 Examples of USB devices 9
Table 1-2 Keys for finding a USB device 14
Table 1-3 Keys for finding a USB interface 15

Chapter 2 **Working With USB Device Interfaces 21**

Listing 2-1 Definitions and global variables 23
Listing 2-2 The main function 24
Listing 2-3 Accessing and programming the raw device 26
Listing 2-4 Releasing the raw device objects 28
Listing 2-5 Configuring a USB device 28
Listing 2-6 Two functions to download firmware to the raw device 29
Listing 2-7 Accessing the bulk test device 30
Listing 2-8 Finding interfaces on the bulk test device 32
Listing 2-9 Two asynchronous I/O completion functions 36

Introduction to USB Device Interface Guide

Note: This document was previously titled *Working With USB Device Interfaces*.

The I/O Kit provides a device interface mechanism that allows applications to communicate with and control hardware from outside the kernel. This document focuses on how to use that mechanism to create an application that detects the attachment of a USB device, communicates with it, and detects its detachment.

This document does not describe how to develop an in-kernel driver for a USB modem or networking device. If you need to do this, refer to the documentation and sample code listed in "See Also" (page 7).

Organization of This Document

This document contains the following chapters:

- "USB Device Overview" (page 9) provides an overview of USB device architecture and terminology and describes how USB devices are represented in Mac OS X.
- "Working With USB Device Interfaces" (page 21) describes how to use the device interface mechanism to create a command-line tool that accesses a USB device.
- "Document Revision History" (page 39) lists the revisions of this document.

See Also

The ADC Reference Library contains several documents on device driver development for Mac OS X and numerous sample drivers and applications.

- *Accessing Hardware From Applications* describes various ways to access devices from outside the kernel, including the device interface mechanism provided by the I/O Kit. For an overview of the I/O Kit terms and concepts used in this document, read the chapter Device Access and the I/O Kit.
- *I/O Kit Framework Reference* contains API reference for I/O Kit methods and functions and for specific device families.
- Sample Code > Hardware & Drivers > USB includes both application-level and in-kernel code samples. Of particular relevance to this document is the application-level sample *USBPrivateDataSample*.
- Additional application-level code samples are included as part of the Mac OS X Developer Tools installation package in `/Developer/Examples/IOKit/usb`.
- Mac OS X Man Pages provides access to existing reference documentation for BSD and POSIX functions and tools in a convenient HTML format.

- The usb mailing list provides a forum for discussing technical issues relating to USB devices in Mac OS X.

If you need to develop an in-kernel driver for a USB modem or networking device, refer to the following:

- *I/O Kit Fundamentals* describes the architecture of the I/O Kit, the object-oriented framework for developing Mac OS X device drivers.
- ADC members can view the AppleUSBCDCDriver project in the source code for Mac OS X v10.3.7 and later, available at Darwin Releases. To find the source code, select a version of Mac OS X equal to or greater than v10.3.7 and click Source (choose the source for the PPC version, if there's a choice). This displays a new page, which lists the open source projects available for the version of Mac OS X you've chosen. Scroll down to AppleUSBCDCDriver and click it to view the source. Be prepared to supply your ADC member name and password.
- Additional code samples that demonstrate specific in-kernel driver programming techniques are included as part of the Mac OS X Developer Tools installation package in `/Developer/Examples/Kernel/IOKit/usb`.

If you're ready to create a universal binary version of your USB device-access application to run in an Intel-based Macintosh, see *Universal Binary Programming Guidelines, Second Edition*. The *Universal Binary Programming Guidelines* describes the differences between the Intel and PowerPC architectures and provides tips for developing a universal binary.

If you are working with a device that complies with the USB mass storage specification but declares its device class to be vendor specific, see *Mass Storage Device Driver Programming Guide* for information on how to ensure the correct built-in driver loads for the device.

Apple provides additional USB information (including the Mac OS X USB Debug Kits) at http://developer.apple.com/hardwaredrivers/usb/index.html.

A detailed description of the USB device specification is beyond the scope of this document—for more information, see *Universal Serial Bus Specification Revision 2.0* available at http://www.usb.org.

USB Device Overview

This chapter provides a summary of USB device architecture and describes how USB devices are represented in Mac OS X. It also presents a few specific guidelines for working with USB devices in an application.For details on the USB specification, see http://www.usb.org.

USB Device Types and Bus Speeds

The USB specification supports a wide selection of devices that range from lower-speed devices such as keyboards, mice, and joysticks to higher-speed devices such as scanners and digital cameras. The specification lists a number of device classes that each define a set of expected device behaviors. Table 1-1 (page 9) lists some examples of USB devices, categorized by class.

Table 1-1 Examples of USB devices

USB device class	USB devices in class
Audio class	Speakers, microphones
Chip Card Interface Device Class	Smart cards, chip cards
Communication class	Speakerphone, modem
Composite class	A device in which all class-specific information is embedded in its interfaces
HID class	Keyboards, mice, joysticks, drawing tablets
Hub class	Hubs provide additional attachment points for USB devices
Mass storage class	Hard drives, flash memory readers, CD Read/Write drives, digital cameras, and high-end media players
Printing class	Printers
Vendor specific	A device that doesn't fit into any other predefined class or one that doesn't use the standard protocols for an existing class
Video class	Digital camcorders, webcams, digital still cameras that support video streaming

Version 1.1 of the USB specification supports two bus speeds:

- Low speed (1.5 Mbps)
- Full speed (12 Mbps)

Version 2.0 of the specification adds another bus speed to this list:

- High speed (480 Mbps)

The USB 2.0 specification is fully compatible with low-speed and full-speed USB devices and even supports the use of cables and connectors made to meet earlier versions of the specification. Apple provides USB 2.0 ports on all new Macintosh computers and fully supports the new specification with Enhanced Host Controller Interface (EHCI) controllers and built-in, low-level USB drivers.

For the most part, you do not have to change existing applications to support the faster data rate because the speed increase and other enhancements are implemented at such a low level. The exceptions to this are some differences in isochronous transfers. For information on how the USB 2.0 specification affects isochronous transfers, see "USB 2.0 and Isochronous Transfers" (page 12).

USB Device Architecture and Terminology

The architecture of a generic USB device is multi-layered. A device consists of one or more configurations, each of which describes a possible setting the device can be programmed into. Such settings can include the power characteristics of the configuration (for example, the maximum power consumed by the configuration and whether it is self-powered or not) and whether the configuration supports remote wake-up.

Each configuration contains one or more interfaces that are accessible after the configuration is set. An interface provides the definitions of the functions available within the device and may even contain alternate settings within a single interface. For example, an interface for an audio device may have different settings you can select for different bandwidths.

Each interface contains zero or more endpoints. An endpoint is a uniquely identifiable portion of a USB device that is the source or sink of information in a communication flow between the host and the device. Each endpoint has characteristics that describe the communication it supports, such as transfer type (control, isochronous, interrupt, or bulk, described in "USB Transfer Types" (page 11)), maximum packet size, and transfer direction (input or output).

Communication with a USB device is accomplished through a pipe, a logical association between an endpoint and software running on the host. Endpoint and pipe are often used synonymously although an endpoint is a component of a USB device and a pipe is a logical abstraction of the communications link between endpoint and host.

USB Device Component Descriptors

Each layer of a USB device provides information about its attributes and resource requirements in its descriptor, a data structure accessible through device interface functions. By examining the descriptors at each layer, you can determine exactly which endpoint you need to communicate successfully with a particular device.

At the top layer is the device descriptor, which has fields associated with information such as the device's class and subclass, vendor and product numbers, and number of configurations. Each configuration in turn has a configuration descriptor containing fields that describe the number of interfaces it supports and the power characteristics of the device when it is in that configuration, along with other information. Each interface supported by a configuration has its own descriptor with fields for information such as the interface class, subclass, and protocol, and the number of endpoints in that interface. At the bottom layer are the endpoint descriptors that specify attributes such as transfer type and maximum packet size.

The USB specification defines a name for each descriptor field, such as the `bDeviceClass` field in the device descriptor and the `bNumInterfaces` field in the configuration descriptor, and each field is associated with a value. For a complete listing of all descriptor fields, see the USB specification at www.usb.org. The USB family defines structures that represent the descriptors defined by the USB specification. For the definitions of these structures, see `USB` in *Kernel Framework Reference*.

USB Composite Class Devices

The USB specification defines a composite class device as a device whose device-descriptor fields for device class (`bDeviceClass`) and device subclass (`bDeviceSubClass`) both have the value `0`. A composite class device appears to the system as a USB device using a single bus address that may present multiple interfaces, each of which represents a separate function. A good example of a composite class device is a multifunction device, such as a device that performs printing, scanning, and faxing. In such a device, each function is represented by a separate interface. In Mac OS X, the I/O Kit loads the `AppleUSBComposite` device driver for composite class devices that do not already have vendor-specific device drivers to drive them. The `AppleUSBComposite` driver configures the device and causes drivers to be loaded for each USB interface.

Although most multifunction USB devices are composite class devices, not all composite class devices are multifunction devices. The manufacturer of a single-function USB device is at liberty to classify the device as a composite class device as long as the device meets the USB specifications. For more information on how Mac OS X represents USB devices and interfaces, see "USB Devices on Mac OS X" (page 13).

USB Transfer Types

The USB specification defines four types of pipe transfer:

- **Control**—intended to support configuration, command, and status communication between the host software and the device. Control transfers support error detection and retry.
- **Interrupt**—used to support small, limited-latency transfers to or from a device such as coordinates from a pointing device or status changes from a modem. Interrupt transfers support error detection and retry.
- **Isochronous**—used for periodic, continuous communication between the host and the device, usually involving time-relevant information such as audio or video data streams. Isochronous transfers do not support error detection or retry.
- **Bulk**—intended for non-periodic, large-packet communication with relaxed timing constraints such as between the host software and a printer or scanner. Bulk transfers support error detection and retry.

Pipes also have a transfer direction associated with them. A control pipe can support bidirectional communication but all other pipes are strictly uni-directional. Therefore, two-way communication requires two pipes, one for input and one for output.

Every USB device is required to implement a default control pipe that provides access to the device's configuration, status, and control information. This pipe, implemented in the `IOUSBDevice` nub object (described in "USB Devices on Mac OS X" (page 13)), is used when a driver such as the `AppleUSBComposite` driver configures the device or when device-specific control and status information is needed. For example, your application would use the default control pipe if it needs to set or choose a configuration for the device. The default control pipe is connected to the default endpoint (endpoint 0). Note that endpoint 0 does not provide an endpoint descriptor and it is never counted in the total number of endpoints in an interface.

The interfaces associated with a configuration can contain any combination of the three remaining pipe types (interrupt, isochronous, and bulk), implemented in the `IOUSBInterface` nub objects (described in "USB Devices on Mac OS X" (page 13)). Your application can query the interface descriptors of a device to select the pipe most suited to its needs.

Stalls and Halts

Although a stall and a halt are different, they are closely related in their effect on data transmission. Halt is a feature of an endpoint and it can be set by either the host or the device itself in response to an error. A stall is a type of handshake packet an endpoint returns when it is unable to transmit or receive data or when its halt feature is set (the host never sends a stall packet). When an endpoint sends a stall packet, the host can halt the endpoint.

Depending on the precise circumstances and on how compliant the device is, the halt feature must be cleared in the host, the endpoint, or both before data transmission can resume. When the halt is cleared the data toggle bit, used to synchronize data transmission, is also reset (see "Data Synchronization in Non-Isochronous Transfers" (page 12) for more information about the data toggle). For information on how to handle these conditions in your application, see "Handling Stalls, Halts, and Data Toggle Resynchronization" (page 16).

Data Synchronization in Non-Isochronous Transfers

The USB specification defines a simple protocol to provide data synchronization across multiple packets for non-isochronous transfers (recall that isochronous transfers do not support error recovery or retry). The protocol is implemented by means of a data toggle bit in both the host and the endpoint which is synchronized at the start of a transaction (or when a reset occurs). The precise synchronization mechanism varies with the type of transfer; see the USB specification for details.

Both the host and the endpoint begin a transaction with their data toggle bits set to zero. In general, the entity receiving data toggles its data toggle bit when it is able to accept the data and it receives an error-free data packet with the correct identification. The entity sending the data toggles its data toggle bit when it receives a positive acknowledgement from the receiver. In this way, the data toggle bits stay synchronized until, for example, a packet with an incorrect identification is received. When this happens, the receiver ignores the packet and does not increment its data toggle bit. When the data toggle bits get out of synchronization (for this or any other reason), you will probably notice that alternate transactions are not getting through in your application. The solution to this is to resynchronize the data toggle bits. For information on how to do this, see "Handling Stalls, Halts, and Data Toggle Resynchronization" (page 16).

USB 2.0 and Isochronous Transfers

The USB 2.0 specification supports the same four transfer types as earlier versions of the specification. In addition to supporting a higher transfer rate, the new specification defines an improved protocol for high-speed transfers and new ways of handling transactions for low-speed and full-speed devices. For details on the protocols and transaction-handling methods, see the specification at http://www.usb.org.

For the most part, these enhancements are implemented at the host software level and do not require changes to your code. For isochronous transfers, however, you should be aware of the following differences:

- Earlier versions of the specification divide bus time into 1-millisecond frames, each of which can carry multiple transactions to multiple destinations. (A transaction contains two or more packets: a token packet and one or more data packets, a handshake packet, or both.) The USB 2.0 specification divides the 1-millisecond frame into eight, 125-microsecond microframes, each of which can carry multiple transactions to multiple destinations.
- The maximum amount of data allowed in a transaction is increased to 3 KB.
- Any isochronous endpoints in a device's default interface must have a maximum packet size of zero. (This means that the default setting for an interface containing isochronous pipes is alternate setting zero and the maximum packet size for that interface's isochronous endpoints must be zero.) This ensures that the host can configure the device no matter how busy the bus is.

For a summary of how these differences affect the Mac OS X USB API, see "Changes in Isochronous Functions to Support USB 2.0" (page 18).

USB Devices on Mac OS X

When a USB device is plugged in, the Mac OS X USB family abstracts the contents of the device descriptor into an I/O Kit nub object called an `IOUSBDevice`. This nub object is attached to the `IOService` plane of the I/O Registry as a child of the driver for the USB controller. The `IOUSBDevice` nub object is then registered for matching with the I/O Kit.

If the device is a composite class device with no vendor-specific driver to match against it, the `AppleUSBComposite` driver matches against it and starts as its provider. The `AppleUSBComposite` driver then configures the device by setting the configuration in the device's list of configuration descriptors with the maximum power usage that can be satisfied by the port to which the device is attached. This allows a device with a low power and a high power configuration to be configured differently depending on whether it's attached to a bus-powered hub or a self-powered hub. In addition, if the `IOUSBDevice` nub object has the "Preferred Configuration" property, the `AppleUSBComposite` driver will always use that value when it attempts to configure the device.

The configuration of the device causes the USB family to abstract each interface descriptor in the chosen configuration into an `IOUSBInterface` nub object. These nub objects are attached to the I/O Registry as children of the original `IOUSBDevice` nub object and are registered for matching with the I/O Kit.

Important: Because a composite class device is configured by the `AppleUSBComposite` driver, setting the configuration again from your application will result in the destruction of the `IOUSBInterface` nub objects and the creation of new ones. In general, the only reason to set the configuration of a composite class device that's matched by the `AppleUSBComposite` driver is to choose a configuration other than the first one.

For non-composite class devices or composite class devices with vendor-specific drivers that match against them, there is no guarantee that any configuration will be set and you may have to perform this task within your application.

It's important to be mindful of the difference between a USB device (represented in the I/O Registry by an `IOUSBDevice` nub object) and its interfaces (each represented by an `IOUSBInterface` nub object). A multifunction USB device, for example, is represented in the I/O Registry by one `IOUSBDevice` object and one `IOUSBInterface` object for each interface.

The distinction between interface and device is important because it determines which object your application must find in the I/O Registry and which type of device interface to get. For example, if your application needs to communicate with a specific interface in a multifunction USB device, it must find that interface and get an `IOUSBInterfaceInterface` to communicate with it. An application that needs to communicate with the USB device as a whole, on the other hand, would need to find the device in the I/O Registry and get an `IOUSBDeviceInterface` to communicate with it. For more information on finding devices and interfaces in the I/O Registry, see "Finding USB Devices and Interfaces" (page 14); for more information on how to get the proper device interface to communicate with a device or interface, see "Using USB Device Interfaces" (page 21).

Finding USB Devices and Interfaces

To find a USB device or interface, use the keys defined in the *Universal Serial Bus Common Class Specification, Revision 1.0* (available for download from http://www.usb.org/developers/devclass_docs/usbccs10.pdf) to create a matching dictionary that defines a particular search. If you are unfamiliar with the concept of device matching, see the section "Finding Devices in the I/O Registry" in *Accessing Hardware From Applications*.

The keys defined in the specification are listed in the tables below. Each key consists of a specific combination of elements in a device or interface descriptor. In the tables below, the elements in a key are separated by the '+' character to emphasize the requirement that all a key's elements must appear together in your matching dictionary. Both tables present the keys in order of specificity: the first key in each table defines the most specific search and the last key defines the broadest search.

Before you build a matching dictionary, be sure you know whether your application needs to communicate with a device or a specific interface in a device. It's especially important to be aware of this distinction when working with multifunction devices. A multifunction device is often a composite class device that defines a separate interface for each function. If, for example, your application needs to communicate with the scanning function of a device that does scanning, faxing, and printing, you need to build a dictionary to match on only the scanning interface (an IOUSBInterface object), not the device as a whole (an IOUSBDevice object). In this situation, you would use the keys defined for interface matching (those shown in Table 1-3 (page 15)), not the keys for device matching.

Table 1-2 (page 14) lists the keys you can use to find devices (not interfaces). Each key element is a piece of information contained in the device descriptor for a USB device.

Table 1-2 Keys for finding a USB device

Key	Notes
`idVendor` + `idProduct` + `bcdDevice`	`bcdDevice` contains the release number of the device
`idVendor` + `idProduct`	
`idVendor` + `bDeviceSubClass` + `bDeviceProtocol`	Use this key only if the device's `bDeviceClass` is $FF
`idVendor` + `bDeviceSubClass`	Use this key only if the device's `bDeviceClass` is $FF
`bDeviceClass` + `bDeviceSubClass` + `bDeviceProtocol`	Use this key only if the device's `bDeviceClass` is *not* $FF
`bDeviceClass` + `bDeviceSubClass`	Use this key only if the device's `bDeviceClass` is *not* $FF

Table 1-3 (page 15) lists the keys you can use to find interfaces (not devices). Each key element is a piece of information contained in an interface descriptor for a USB device.

Table 1-3 Keys for finding a USB interface

Key	Notes
`idVendor` + `idProduct` + `bcdDevice` + `bConfigurationValue` + `bInterfaceNumber`	
`idVendor` + `idProduct` + `bConfigurationValue` + `bInterfaceNumber`	
`idVendor` + `bInterfaceSubClass` + `bInterfaceProtocol`	Use this key only if `bInterfaceClass` is $FF
`idVendor` + `bInterfaceSubClass`	Use this key only if `bInterfaceSubClass` is $FF
`bInterfaceClass` + `bInterfaceSubClass` + `bInterfaceProtocol`	Use this key only if `bInterfaceSubClass` is *not* $FF
`bInterfaceClass` + `bInterfaceSubClass`	Use this key only if `bInterfaceSubClass` is *not* $FF

For a successful search, you must add the elements of exactly one key to your matching dictionary. If your matching dictionary contains a combination of elements not defined by any key, the search will be unsuccessful. For example, if you create a matching dictionary containing values representing a device's vendor, product, and protocol, the search will be unsuccessful even if a device with those precise values in its device descriptor is currently represented by an `IOUSBDevice` nub in the I/O Registry. This is because there is no key in Table 1-2 (page 14) that combines the `idVendor`, `idProduct`, and `bDeviceProtocol` elements.

USB Family Error Codes

As you develop an application to access a USB device or interface, you will probably encounter error codes specific to the Mac OS X USB family. If you are using Xcode, you can search for information about these error codes in the Xcode documentation window.

To find error code documentation, select Documentation from the Xcode Help menu. Select Full-Text Search from the pull-down menu associated with the search field (click the magnifying glass icon to reveal the menu). Select Reference Library in the Search Groups pane at the left of the window. Type an error code number in the search field, such as 0xe0004057, and press Return. Select the most relevant entry in the search results to display the document in the lower portion of the window. Use the Find command (press Command-F) to find the error code in this document. Using the example of error code 0xe0004057, you'll see that this error is returned when the endpoint has not been found.

For help with deciphering I/O Kit error codes in general, see Technical Q&A QA1075, "Making sense of I/O Kit error codes."

Determining Which Interface Version to Use

As described in "USB Devices on Mac OS X" (page 13), the Mac OS X USB family provides an `IOUSBDeviceInterface` object you use to communicate with a USB device as a whole and an `IOUSBInterfaceInterface` object you use to communicate with an interface in a USB device. There are a number of different versions of the USB family, however, some of which provide new versions of these interface objects. (One way to find the version of the USB family installed in your computer is to view the Finder preview information for the `IOUSBFamily.kext` located in `/System/Library/Extensions`.) This section describes how to make sure you use the correct interface object and how to view the documentation for the interface objects.

The first version of the USB family was introduced in Mac OS X v10.0 and contains the first versions of the interface objects `IOUSBDeviceInterface` and `IOUSBInterfaceInterface`. When new versions of the USB family introduce new functions for an interface object, a new version of the interface object is created, which gives access to both the new functions and all functions defined in all previous versions of that interface object. For example, the `IOUSBDeviceInterface197` object provides two new functions you can use with version 1.9.7 of the USB family (available in Mac OS X v10.2.3 and later), in addition to all functions available in the previous device interface objects `IOUSBDeviceInterface187`, `IOUSBDeviceInterface182`, and `IOUSBDeviceInterface`.

As you develop an application that accesses a USB device or interface, you should use the latest version of the interface object that is available in the earliest version of Mac OS X that you want to support. For example, if your application must run in Mac OS X v10.0, you must use the `IOUSBDeviceInterface` and `IOUSBInterfaceInterface` objects. If, however, you develop an application to run in Mac OS X v10.4 and later, you use the `IOUSBDeviceInterface197` object to access the device as a whole and the `IOUSBInterfaceInterface220` object to access an interface in it. This is because `IOUSBDeviceInterface197` is available in Mac OS X version 10.2.3 and later and `IOUSBInterfaceInterface220` is available in Mac OS X v10.4 and later.

Note: When you view the documentation for these interface objects, notice that each version is documented separately. For example, the documentation for `IOUSBDeviceInterface197` contains information about the two new functions introduced in this version, but does not repeat the documentation for the functions introduced in `IOUSBDeviceInterface187`, `IOUSBDeviceInterface182`, and `IOUSBDeviceInterface`.

Tasks and Caveats

This section presents some specific tasks your application might need to perform, along with some caveats related to USB 2.0 support of which you should be aware.

Handling Stalls, Halts, and Data Toggle Resynchronization

As described in "Stalls and Halts " (page 12), stalls and halts are closely related in their effect on data transmission. To simplify the API, the USB family uses the pipe stall terminology in the names of the functions that handle these conditions:

- `ClearPipeStall`
- `ClearPipeStallBothEnds`

The `ClearPipeStall` function operates exclusively on the host controller side, clearing the halt feature and resetting the data toggle bit to zero. If the endpoint's halt feature and data toggle bit must be reset as well, your application must do so explicitly, using one of the `ControlRequest` functions to send the appropriate device request. See the documentation for the `USB.h` header file in *I/O Kit Framework Reference* for more information about standard device requests.

In Mac OS X version 10.2 and later, you can use the `ClearPipeStallBothEnds` function which, as its name suggests, clears the halt and resets the data toggle bit on both sides at the same time.

Using the Low Latency Isochronous Functions

In Mac OS X, the time between when an isochronous transaction completes on the USB bus and when you receive your callback can stretch to tens of milliseconds. This is because the callback happens on the USB family work loop, which runs at a lower priority than some other threads in the system. In most cases, you can work around this delay by queuing read and write requests so that the next transaction is scheduled and ready to start before you receive the callback from the current transaction. In fact, this scheme is a good way to achieve higher performance whether or not low latency is a requirement of your application.

In a few cases, however, queuing isochronous transactions to keep the pipe busy is not enough to prevent a latency problem that a user might notice. Consider an application that performs audio processing on some USB input (from a musical instrument, for example) before sending the processed data out to USB speakers. In this scenario, a user hears both the raw, unprocessed output of the instrument and the processed output of the speakers. Of course, some small delay between the time the instrument creates the raw sound waves and the time the speaker emits the processed sound waves is unavoidable. If this delay is greater than about 8 milliseconds, however, the user will notice.

In Mac OS X version 10.2.3 (version 1.9.2 of the USB family) the USB family solves this problem by taking advantage of the predictability of isochronous data transfers. By definition, isochronous mode guarantees the delivery of some amount of data every frame or microframe. In earlier versions of Mac OS X, however, it was not possible to find out the exact amount of data that was transferred by a given time. This meant that an application could not begin processing the data until it received the callback associated with the transaction, telling it the transfer status and the actual amount of data that was transferred.

Version 1.9.2 of the USB family introduced the `LowLatencyReadIsochPipeAsync` and `LowLatencyWriteIsochPipeAsync` functions. These functions update the frame list information (including the transfer status and the number of bytes actually transferred) at primary interrupt time. Using these functions, an application can request that the frame list information be updated as frequently as every millisecond. This means an application can retrieve and begin processing the number of bytes actually transferred once a millisecond, without waiting for the entire transaction to complete.

Important: Because these functions cause processing at primary interrupt time, it is essential you use them only if it is absolutely necessary. Overuse of these functions can cause degradation of system performance.

To support the low latency isochronous read and write functions, the USB family also introduced functions to create and destroy the buffers that hold the frame list information and the data. Although you can choose to create a single data buffer and a single frame list buffer or multiple buffers of each type, you must use the `LowLatencyCreateBuffer` function to create them. Similarly, you must use the `LowLatencyDestroyBuffer` function to destroy the buffers after you are finished with them. This restricts all necessary communication with kernel entities to the USB family.

For reference documentation on the low latency isochronous functions, see the `IOUSBLib.h` documentation in *I/O Kit Framework Reference*.

Errors Reported by the EHCI Hub

The EHCI hub that supports high-speed devices (as well as low-speed and full-speed devices) provides coarser-grained error reporting than the OHCI hub does. For example, with an OHCI hub, you might receive an "endpoint timed out" error if you unplug the device while it is active. If you perform the same action with an EHCI hub, you might receive a "pipe stalled" error instead.

The Apple EHCI hub driver cannot get more detailed error information from the hub, so it alternates between reporting "device not responding" and "pipe stalled" regardless of the actual error reported by the device. To avoid problems with your code, be sure your application does not rely on other, more specific errors to make important decisions.

Changes in Isochronous Functions to Support USB 2.0

Recall that the USB 2.0 specification divides the 1-millisecond frame into eight, 125-microsecond microframes. The USB family handles this by reinterpreting some function parameters (where appropriate) and adding a couple of new functions. This section summarizes these changes; for reference documentation, see documentation for `IOUSBLib.h` in *I/O Kit Framework Reference*.

The functions you use to read from and write to isochronous endpoints are `ReadIsochPipeAsync` and `WriteIsochPipeAsync`. Both functions include the following two parameters:

- *`numFrames`*—The number of frames for which to transfer data
- *`frameList`*—A pointer to an array of structures that describe the frames

If you need to handle high-speed isochronous transfers, you can think of these parameters as referring to "transfer opportunities" instead of frames. In other words, *`numFrames`* can refer to a number of frames for full-speed devices or to a number of microframes for high-speed devices. Similarly, *`frameList`* specifies the list of transfers you want to occur, whether they are in terms of frames or microframes.

Note: The `ReadIsochPipeAsync` and `WriteIsochPipeAsync` functions also have the *`frameStart`* parameter in common, but it does not get reinterpreted. This is because all isochronous transactions, including high-speed isochronous transactions, start on a frame boundary, not a microframe boundary.

To help you determine whether a device is functioning in full-speed or high-speed mode, the USB family added the `GetFrameListTime` function, which returns the number of microseconds in a frame. By examining the result (`kUSBFullSpeedMicrosecondsInFrame` or `kUSBHighSpeedMicrosecondsInFrame`) you can tell in which mode the device is operating.

The USB family also added the `GetBusMicroFrameNumber` function which is similar to the `GetBusFrameNumber` function, except that it returns both the current frame and microframe number and includes the time at which that information was retrieved.

To handle the new specification's requirement that isochronous endpoints in a device's default interface have a maximum packet size of zero, the USB family added functions that allow you to balance bandwidth allocations among isochronous endpoints. A typical scenario is this:

1. Call `GetBandwidthAvailable` (available in Mac OS X version 10.2 and later) to determine how much bandwidth is currently available for allocation to isochronous endpoints.
2. Call `GetEndpointProperties` (available in Mac OS X version 10.2 and later) to examine the alternate settings of an interface and find one that uses an appropriate amount of bandwidth.
3. Call `SetAlternateInterface` (available in Mac OS X version 10.0 and later) to create the desired interface and allocate the pipe objects.
4. Call `GetPipeProperties` (available in Mac OS X version 10.0 and later) on the chosen isochronous endpoint. This is a very important step because `SetAlternateInterface` will succeed, even if there is not enough bandwidth for the endpoints. Also, another device might have claimed the bandwidth that was available at the time the `GetBandwidthAvailable` function returned. If this happens, the maximum packet size for your chosen endpoint (contained in the `maxPacketSize` field) is now zero, which means that the bandwidth is no longer available.

In addition, in Mac OS X version 10.2, the USB family added the `SetPipePolicy` function, which allows you to relinquish bandwidth that might have been specified in an alternate setting.

USB Device Access in an Intel-Based Macintosh

This section provides an overview of some of the issues related to developing a universal binary version of an application that accesses a USB device. Before you read this section, be sure to read *Universal Binary Programming Guidelines, Second Edition*. That document covers architectural differences and byte-ordering formats and provides comprehensive guidelines for code modification and building universal binaries. The guidelines in that document apply to all types of applications, including those that access hardware.

Before you build your application as a universal binary, make sure that:

- You port your project to GCC 4 (Xcode uses GCC 4 to target Intel-based Macintosh computers)
- You install the Mac OS X v10.4 universal SDK
- You develop your project in Xcode 2.1 or later

The USB bus is a little-endian bus. Structured data appears on the bus in the little-endian format regardless of the native endian format of the computer an application is running in. If you've developed a USB device-access application to run in a PowerPC-based Macintosh, you probably perform some byte swapping on data you read from the USB bus because the PowerPC processor uses the big-endian format. For example, the USB configuration descriptor structure contains a two-byte field that holds the descriptor length. If your PowerPC application reads this structure from the USB bus (instead of receiving it from a USB device interface function), you need to swap the value from the USB bus format (little endian) to the PowerPC format (big endian).

The USB family provides several swapping macros that swap from USB to host and from host to USB (for more information on these macros, see USB.h). The Kernel framework also provides byte-swapping macros and functions you can use in high-level applications (see the `OSByteOrder.h` header file in `libkern`). If you use these macros in your application, you shouldn't have any trouble developing a universal binary version of your application. This is because these macros determine at compile time if a swap is necessary. If, however, your application uses hard-coded swaps from little endian to big endian, your application will

not run correctly in an Intel-based Macintosh. As you develop a universal binary version of your application, therefore, be sure to use the USB family swapping macros or the macros in `libkern/OSByteOrder.h` for all byte swapping.

Although you may need to perform byte swapping on values your application reads from the USB bus, you do not need to perform any byte swapping on values you pass in arguments to functions in the USB family API. You should pass argument values in the computer's host format. Likewise, any values you receive from the USB family functions will be in the computer's host format.

Working With USB Device Interfaces

This chapter describes how to develop a user-space tool that finds and communicates with an attached USB device and one of its interfaces. The sample code in this chapter is taken from the USB Notification Example sample project, available in `/Developer/Examples/IOKit/usb` of the Developer version of Mac OS X.

Important: The sample code featured in this document is intended to illustrate how to access a USB device from an application. It is not intended to provide guidance on error handling and other features required for production-quality code.

Using USB Device Interfaces

Applications running in Mac OS X get access to USB devices by using I/O Kit functions to acquire a device interface, a type of plug-in that specifies functions the application can call to communicate with the device. The USB family provides two types of device interface:

- `IOUSBDeviceInterface` for communicating with the device itself
- `IOUSBInterfaceInterface` for communicating with an interface in the device

Both device interfaces are defined in `/System/Library/Frameworks/IOKit.framework/Headers/usb/IOUSBLib.h`.

Communicating with the device itself is usually only necessary when you need to set or change its configuration. For example, vendor-specific devices are often not configured because there are no default drivers that set a particular configuration. In this case, your application must use the device interface for the device to set the configuration it needs so the interfaces become available.

The process of finding and communicating with a USB device is divided into two sets of steps. The first set outlines how to find a USB device, acquire a device interface of type `IOUSBDeviceInterface` for it, and set or change its configuration. The second set describes how to find an interface in a device, acquire a device interface of type `IOUSBInterfaceInterface` for it, and use it to communicate with that interface. If you need to communicate with an unconfigured device or if you need to change a device's configuration, you follow both sets of steps. If you need to communicate with a device that is already configured to your specification, you follow only the second set of steps. The sample code in "Accessing a USB Device" (page 22) follows both sets of steps and extends them to include setting up notifications it can receive when devices are dynamically added or removed.

Follow this first set of steps *only* to set or change the configuration of a device. If the device you're interested in is already configured for your needs, skip these steps and follow the second set of steps.

1. Find the `IOUSBDevice` object that represents the device in the I/O Registry. This includes setting up a matching dictionary with a key from the *USB Common Class Specification* (see "Finding USB Devices and Interfaces" (page 14)). The sample code uses the key elements `kUSBVendorName` and `kUSBProductName` to find a particular USB device (this is the second key listed in Table 1-2 (page 14)).

2. Create a device interface of type `IOUSBDeviceInterface` for the device. This device interface provides functions that perform tasks such as setting or changing the configuration of the device, getting information about the device, and resetting the device.

3. Examine the device's configurations with `GetConfigurationDescriptorPtr`, choose the appropriate one, and call `SetConfiguration` to set the device's configuration and instantiate the `IOUSBInterface` objects for that configuration.

Follow this second set of steps to find and choose an interface, acquire a device interface for it, and communicate with the device.

1. Create an interface iterator to iterate over the available interfaces.

2. Create a device interface for each interface so you can examine its properties and select the appropriate one. To do this, you create a device interface of type `IOUSBInterfaceInterface`. This device interface provides functions that perform tasks such as getting information about the interface, setting the interface's alternate setting, and accessing its pipes.

3. Use the `USBInterfaceOpen` function to open the selected interface. This will cause the pipes associated with the interface to be instantiated so you can examine the properties of each and select the appropriate one.

4. Communicate with the device through the selected pipe. You can write to and read from the pipe synchronously or asynchronously—the sample code in "Accessing a USB Device" (page 22) shows how to do both.

Accessing a USB Device

This section provides snippets of sample code that show how to access a Cypress EZ-USB chip with an 8051 microcontroller core. The sample code follows the first set of steps in section "Using USB Device Interfaces" (page 21) to find the Cypress EZ-USB chip in its default, unprogrammed state (also referred to as the "raw device"). It then configures the device and downloads firmware provided by Cypress to program the chip to behave as a device that echoes all information it receives on its bulk out pipe to its bulk in pipe.

Once the chip has been programmed, the device nub representing the default, unprogrammed device is detached from the I/O Registry and a new device nub, representing the programmed chip, is attached. To communicate with the programmed chip (also referred to as the "bulk test device"), the sample code must perform the first set of steps again to find the device, create a device interface for it, and configure it. Then it performs the second set of steps to find an interface, create a device interface for it, and test the device. The sample code also shows how to set up notifications for the dynamic addition and removal of a device.

If you want to build and run the sample code, open the USB Notification Example project (located in `/Developer/Examples/IOKit/usb`) in Xcode. You can observe the resulting tool's status messages in the Xcode Run Log and terminate the tool by clicking the Terminate button. Alternatively, you can open a Terminal window, navigate to the location of the executable (for example, `/Developer/Examples/IOKit/usb/USBNotification Example/build`), run the tool, and terminate it by pressing Control-C. Note that the project also includes the `bulktest.c` file, which contains the firmware to download to the raw device, and the `hex2c.h` file, which defines the structure the firmware uses.

Definitions and Global Variables

The code in the USB Notification Example uses the definitions and global variables shown in Listing 2-1 (page 23). The definition of `USE_ASYNC_IO` allows you to choose to use either synchronous or asynchronous calls to read from and write to the chip by commenting out the line or leaving it in, respectively. The definition of `kTestMessage` sets up a simple message to write to the device. The remaining definitions are specific to the Cypress EZ-USB chip.

Listing 2-1 Definitions and global variables

```
#define USE_ASYNC_IO    //Comment this line out if you want to use
                        //synchronous calls for reads and writes
#define kTestMessage        "Bulk I/O Test"
#define k8051_USBCS         0x7f92
#define kOurVendorID        1351    //Vendor ID of the USB device
#define kOurProductID           8193    //Product ID of device BEFORE it
                                        //is programmed (raw device)
#define kOurProductIDBulkTest   4098    //Product ID of device AFTER it is
                                        //programmed (bulk test device)

//Global variables
static IONotificationPortRef    gNotifyPort;
static io_iterator_t            gRawAddedIter;
static io_iterator_t            gRawRemovedIter;
static io_iterator_t            gBulkTestAddedIter;
static io_iterator_t            gBulkTestRemovedIter;
static char                     gBuffer[64];
```

The main Function

The `main` function in the USB Notification Example project (contained in the file `main.c`) accomplishes the following tasks.

- It establishes communication with the I/O Kit and sets up a matching dictionary to find the Cypress EZ-USB chip.
- It sets up an asynchronous notification to be called when an unprogrammed (raw) device is first attached to the I/O Registry and another to be called when the device is removed.
- It modifies the matching dictionary to find the programmed (bulk test) device.
- It sets up additional notifications to be called when the bulk test device is first attached or removed.
- It starts the run loop so the notifications that have been set up will be received.

The `main` function uses I/O Kit functions to set up and modify a matching dictionary and set up notifications, and Core Foundation functions to set up the run loop for receiving the notifications. It calls the following functions to access both the raw device and the bulk test device.

- `RawDeviceAdded`, shown in Listing 2-3 (page 26), iterates over the set of matching devices and creates a device interface for each one. It calls `ConfigureDevice` (shown in Listing 2-5 (page 28)) to set the device's configuration, and then `DownloadToDevice` (shown in Listing 2-6 (page 29)) to download the firmware to program it.

- `RawDeviceRemoved`, shown in Listing 2-4 (page 28), iterates over the set of matching devices and releases each one in turn.
- `BulkTestDeviceAdded`, shown in Listing 2-7 (page 30), iterates over the new set of matching devices, creates a device interface for each one, and calls `ConfigureDevice` (shown in Listing 2-5 (page 28)) to set the device's configuration. It then calls `FindInterfaces` (shown in Listing 2-8 (page 32)) to get access to the interfaces on the device.
- `BulkTestDeviceRemoved` iterates over the new set of matching devices and releases each one in turn. This function is not shown in this chapter; see `RawDeviceRemoved` (Listing 2-4 (page 28)) for a nearly identical function.

Listing 2-2 The main function

```
int main (int argc, const char *argv[])
{
    mach_port_t             masterPort;
    CFMutableDictionaryRef  matchingDict;
    CFRunLoopSourceRef      runLoopSource;
    kern_return_t           kr;
    SInt32                  usbVendor = kOurVendorID;
    SInt32                  usbProduct = kOurProductID;

    // Get command line arguments, if any
    if (argc > 1)
        usbVendor = atoi(argv[1]);
    if (argc > 2)
        usbProduct = atoi(argv[2]);

    //Create a master port for communication with the I/O Kit
    kr = IOMasterPort(MACH_PORT_NULL, &masterPort);
    if (kr || !masterPort)
    {
        printf("ERR: Couldn't create a master I/O Kit port(%08x)\n", kr);
        return -1;
    }
    //Set up matching dictionary for class IOUSBDevice and its subclasses
    matchingDict = IOServiceMatching(kIOUSBDeviceClassName);
    if (!matchingDict)
    {
        printf("Couldn't create a USB matching dictionary\n");
        mach_port_deallocate(mach_task_self(), masterPort);
        return -1;
    }

    //Add the vendor and product IDs to the matching dictionary.
    //This is the second key in the table of device-matching keys of the
    //USB Common Class Specification
    CFDictionarySetValue(matchingDict, CFSTR(kUSBVendorName),
                         CFNumberCreate(kCFAllocatorDefault,
                                        kCFNumberSInt32Type, &usbVendor));
    CFDictionarySetValue(matchingDict, CFSTR(kUSBProductName),
                         CFNumberCreate(kCFAllocatorDefault,
                                        kCFNumberSInt32Type, &usbProduct));

    //To set up asynchronous notifications, create a notification port and
    //add its run loop event source to the program's run loop
```

```
gNotifyPort = IONotificationPortCreate(masterPort);
runLoopSource = IONotificationPortGetRunLoopSource(gNotifyPort);
CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource,
                   kCFRunLoopDefaultMode);

//Retain additional dictionary references because each call to
//IOServiceAddMatchingNotification consumes one reference
matchingDict = (CFMutableDictionaryRef) CFRetain(matchingDict);
matchingDict = (CFMutableDictionaryRef) CFRetain(matchingDict);
matchingDict = (CFMutableDictionaryRef) CFRetain(matchingDict);

//Now set up two notifications: one to be called when a raw device
//is first matched by the I/O Kit and another to be called when the
//device is terminated
//Notification of first match:
kr = IOServiceAddMatchingNotification(gNotifyPort,
                kIOFirstMatchNotification, matchingDict,
                RawDeviceAdded, NULL, &gRawAddedIter);
//Iterate over set of matching devices to access already-present devices
//and to arm the notification
RawDeviceAdded(NULL, gRawAddedIter);

//Notification of termination:
kr = IOServiceAddMatchingNotification(gNotifyPort,
                kIOTerminatedNotification, matchingDict,
                RawDeviceRemoved, NULL, &gRawRemovedIter);
//Iterate over set of matching devices to release each one and to
//arm the notification
RawDeviceRemoved(NULL, gRawRemovedIter);

//Now change the USB product ID in the matching dictionary to match
//the one the device will have after the firmware has been downloaded
usbProduct = kOurProductIDBulkTest;
CFDictionarySetValue(matchingDict, CFSTR(kUSBProductName),
                   CFNumberCreate(kCFAllocatorDefault,
                              kCFNumberSInt32Type, &usbProduct));

//Now set up two notifications: one to be called when a bulk test device
//is first matched by the I/O Kit and another to be called when the
//device is terminated.
//Notification of first match
kr = IOServiceAddMatchingNotification(gNotifyPort,
                kIOFirstMatchNotification, matchingDict,
                BulkTestDeviceAdded, NULL, &gBulkTestAddedIter);
//Iterate over set of matching devices to access already-present devices
//and to arm the notification
BulkTestDeviceAdded(NULL, gBulkTestAddedIter);

//Notification of termination
kr = IOServiceAddMatchingNotification(gNotifyPort,
                kIOTerminatedNotification, matchingDict,
                BulkTestDeviceRemoved, NULL, &gBulkTestRemovedIter);
//Iterate over set of matching devices to release each one and to
//arm the notification. NOTE: this function is not shown in this document.
BulkTestDeviceRemoved(NULL, gBulkTestRemovedIter);

//Finished with master port
mach_port_deallocate(mach_task_self(), masterPort);
```

```
    masterPort = 0;

    //Start the run loop so notifications will be received
    CFRunLoopRun();

    //Because the run loop will run forever until interrupted,
    //the program should never reach this point
    return 0;
}
```

Working With the Raw Device

Now that you've obtained an iterator for a set of matching devices, you can use it to gain access to each raw device, configure it, and download the appropriate firmware to it. The function `RawDeviceAdded` (shown in Listing 2-3 (page 26)) uses I/O Kit functions to create a device interface for each device and then calls the following functions to configure the device and download firmware to it.

- `ConfigureDevice`, shown in Listing 2-5 (page 28), uses device interface functions to get the number of configurations, examine the first one, and set the device's configuration.
- `DownloadToDevice`, shown in Listing 2-6 (page 29), downloads the firmware in `bulktest.c` to the device.

Listing 2-3 Accessing and programming the raw device

```
void RawDeviceAdded(void *refCon, io_iterator_t iterator)
{
    kern_return_t               kr;
    io_service_t                usbDevice;
    IOCFPlugInInterface         **plugInInterface = NULL;
    IOUSBDeviceInterface        **dev = NULL;
    HRESULT                     result;
    SInt32                      score;
    UInt16                      vendor;
    UInt16                      product;
    UInt16                      release;

    while (usbDevice = IOIteratorNext(iterator))
    {
        //Create an intermediate plug-in
        kr = IOCreatePlugInInterfaceForService(usbDevice,
                    kIOUSBDeviceUserClientTypeID, kIOCFPlugInInterfaceID,
                    &plugInInterface, &score);
        //Don't need the device object after intermediate plug-in is created
        kr = IOObjectRelease(usbDevice);
        if ((kIOReturnSuccess != kr) || !plugInInterface)
        {
            printf("Unable to create a plug-in (%08x)\n", kr);
            continue;
        }
        //Now create the device interface
        result = (*plugInInterface)->QueryInterface(plugInInterface,
                        CFUUIDGetUUIDBytes(kIOUSBDeviceInterfaceID),
                        (LPVOID *)&dev);
        //Don't need the intermediate plug-in after device interface
```

```
        //is created
        (*plugInInterface)->Release(plugInInterface);

        if (result || !dev)
        {
            printf("Couldn't create a device interface (%08x)\n",
                                                    (int) result);
            continue;
        }

        //Check these values for confirmation
        kr = (*dev)->GetDeviceVendor(dev, &vendor);
        kr = (*dev)->GetDeviceProduct(dev, &product);
        kr = (*dev)->GetDeviceReleaseNumber(dev, &release);
        if ((vendor != kOurVendorID) || (product != kOurProductID) ||
            (release != 1))
        {
            printf("Found unwanted device (vendor = %d, product = %d)\n",
                    vendor, product);
            (void) (*dev)->Release(dev);
            continue;
        }

        //Open the device to change its state
        kr = (*dev)->USBDeviceOpen(dev);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to open device: %08x\n", kr);
            (void) (*dev)->Release(dev);
            continue;
        }
        //Configure device
        kr = ConfigureDevice(dev);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to configure device: %08x\n", kr);
            (void) (*dev)->USBDeviceClose(dev);
            (void) (*dev)->Release(dev);
            continue;
        }

        //Download firmware to device
        kr = DownloadToDevice(dev);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to download firmware to device: %08x\n", kr);
            (void) (*dev)->USBDeviceClose(dev);
            (void) (*dev)->Release(dev);
            continue;
        }

        //Close this device and release object
        kr = (*dev)->USBDeviceClose(dev);
        kr = (*dev)->Release(dev);
    }
}
```

The function `RawDeviceRemoved` simply uses the iterator obtained from the `main` function (shown in Listing 2-2 (page 24)) to release each device object. This also has the effect of arming the raw device termination notification so it will notify the program of future device removals. `RawDeviceRemoved` is shown in Listing 2-4 (page 28).

Listing 2-4 Releasing the raw device objects

```
void RawDeviceRemoved(void *refCon, io_iterator_t iterator)
{
    kern_return_t   kr;
    io_service_t    object;

    while (object = IOIteratorNext(iterator))
    {
        kr = IOObjectRelease(object);
        if (kr != kIOReturnSuccess)
        {
            printf("Couldn't release raw device object: %08x\n", kr);
            continue;
        }
    }
}
```

Although every USB device has one or more configurations, unless the device is a composite class device that's been matched by the `AppleUSBComposite` driver which automatically sets the first configuration, none of those configurations may have been set. Therefore, your application may have to use device interface functions to get the appropriate configuration value and use it to set the device's configuration. In the sample code, the function `ConfigureDevice` (shown in Listing 2-5 (page 28)) accomplishes this task. In fact, it is called twice: once by `RawDeviceAdded` to configure the raw device and again by `BulkTestDeviceAdded` (shown in Listing 2-7 (page 30)) to configure the bulk test device.

Listing 2-5 Configuring a USB device

```
IOReturn ConfigureDevice(IOUSBDeviceInterface **dev)
{
    UInt8                           numConfig;
    IOReturn                        kr;
    IOUSBConfigurationDescriptorPtr configDesc;

    //Get the number of configurations. The sample code always chooses
    //the first configuration (at index 0) but your code may need a
    //different one
    kr = (*dev)->GetNumberOfConfigurations(dev, &numConfig);
    if (!numConfig)
        return -1;

    //Get the configuration descriptor for index 0
    kr = (*dev)->GetConfigurationDescriptorPtr(dev, 0, &configDesc);
    if (kr)
    {
        printf("Couldn't get configuration descriptor for index %d (err =
                %08x)\n", 0, kr);
        return -1;
    }

    //Set the device's configuration. The configuration value is found in
    //the bConfigurationValue field of the configuration descriptor
```

```
    kr = (*dev)->SetConfiguration(dev, configDesc->bConfigurationValue);
    if (kr)
    {
        printf("Couldn't set configuration to value %d (err = %08x)\n", 0,
                kr);
        return -1;
    }
    return kIOReturnSuccess;
}
```

Now that the device is configured, you can download firmware to it. Cypress makes firmware available to program the EZ-USB chip to emulate different devices. The sample code in this document uses firmware that programs the chip to be a bulk test device, a device that takes the data it receives from its bulk out pipe and echoes it to its bulk in pipe. The firmware, contained in the file `bulktest.c`, is an array of `INTEL_HEX_RECORD` structures (defined in the file `hex2c.h`).

The function `DownloadToDevice` uses the function `WriteToDevice` (shown together in Listing 2-6 (page 29)) to prepare the device to receive the download and then to write information from each structure to the appropriate address on the device. When all the firmware has been downloaded, `DownloadToDevice` calls `WriteToDevice` a last time to inform the device that the download is complete. At this point, the raw device detaches itself from the bus and reattaches as a bulk test device. This causes the device nub representing the raw device to be removed from the I/O Registry and a new device nub, representing the bulk test device, to be attached.

Listing 2-6 Two functions to download firmware to the raw device

```
IOReturn DownloadToDevice(IOUSBDeviceInterface **dev)
{
    int         i;
    UInt8       writeVal;
    IOReturn    kr;

    //Assert reset. This tells the device that the download is
    //about to occur
    writeVal = 1;   //For this device, a value of 1 indicates a download
    kr = WriteToDevice(dev, k8051_USBCS, 1, &writeVal);
    if (kr != kIOReturnSuccess)
    {
        printf("WriteToDevice reset returned err 0x%x\n", kr);
        (*dev)->USBDeviceClose(dev);
        (*dev)->Release(dev);
        return kr;
    }

    //Download firmware
    i = 0;
    while (bulktest[i].Type == 0)   //While bulktest[i].Type == 0, this is
    {                               //not the last firmware record to
                                    //download
        kr = WriteToDevice(dev, bulktest[i].Address,
                           bulktest[i].Length, bulktest[i].Data);
        if (kr != kIOReturnSuccess)
        {
            printf("WriteToDevice download %i returned err 0x%x\n", i,
                    kr);
            (*dev)->USBDeviceClose(dev);
            (*dev)->Release(dev);
```

```
            return kr;
        }
        i++;
    }

    //De-assert reset. This tells the device that the download is complete
    writeVal = 0;
    kr = WriteToDevice(dev, k8051_USBCS, 1, &writeVal);
    if (kr != kIOReturnSuccess)
        printf("WriteToDevice run returned err 0x%x\n", kr);

    return kr;
}


IOReturn WriteToDevice(IOUSBDeviceInterface **dev, UInt16 deviceAddress,
                       UInt16 length, UInt8 writeBuffer[])
{
    IOUSBDevRequest     request;

    request.bmRequestType = USBmakebmRequestType(kUSBOut, kUSBVendor,
                                                 kUSBDevice);
    request.bRequest = 0xa0;
    request.wValue = deviceAddress;
    request.wIndex = 0;
    request.wLength = length;
    request.pData = writeBuffer;

    return (*dev)->DeviceRequest(dev, &request);
}
```

Working With the Bulk Test Device

After you download the firmware to the device, the raw device is no longer attached to the bus. To gain access to the bulk test device, you repeat most of the same steps you used to get access to the raw device.

- Use the iterator obtained by a call to `IOServiceAddMatchingNotification` in the `main` function (shown in Listing 2-2 (page 24)) to iterate over a set of matching devices.
- Create a device interface for each device.
- Configure the device.

This time, however, the next step is to find the interfaces on the device so you can choose the appropriate one and get access to its pipes. Because of the similarities of these tasks, the function `BulkTestDeviceAdded` follows the same outline of the `RawDeviceAdded` function except that instead of downloading firmware to the device, it calls `FindInterfaces` (shown in Listing 2-8 (page 32)) to examine the available interfaces and their pipes. The code in Listing 2-7 (page 30) replaces most of the `BulkTestDeviceAdded` function's code with comments, focusing on the differences between it and the `RawDeviceAdded` function.

Listing 2-7 Accessing the bulk test device

```
void BulkTestDeviceAdded(void *refCon, io_iterator_t iterator)
{
    kern_return_t           kr;
```

```
    io_service_t            usbDevice;
    IOUSBDeviceInterface    **device=NULL;

    while (usbDevice = IOIteratorNext(iterator))
    {
        //Create an intermediate plug-in using the
        //IOCreatePlugInInterfaceForService function

        //Release the device object after getting the intermediate plug-in

        //Create the device interface using the QueryInterface function

        //Release the intermediate plug-in object

        //Check the vendor, product, and release number values to
        //confirm we've got the right device

        //Open the device before configuring it
        kr = (*device)->USBDeviceOpen(device);

        //Configure the device by calling ConfigureDevice

        //Close the device and release the device interface object if
        //the configuration is unsuccessful

        //Get the interfaces
        kr = FindInterfaces(device);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to find interfaces on device: %08x\n", kr);
            (*device)->USBDeviceClose(device);
            (*device)->Release(device);
            continue;
        }

//If using synchronous IO, close and release the device interface here
#ifndef USB_ASYNC_IO
        kr = (*device)->USBDeviceClose(device);
        kr = (*device)->Release(device);
#endif
    }
}
```

The function `BulkTestDeviceRemoved` simply uses the iterator obtained from the `main` function (shown in Listing 2-2 (page 24)) to release each device object. This also has the effect of arming the bulk test device termination notification so it will notify the program of future device removals.The `BulkTestDeviceRemoved` function is identical to the `RawDeviceRemoved` function (shown in Listing 2-4 (page 28)), with the exception of the wording of the printed error statement.

Working With Interfaces

Now that you've configured the device, you have access to its interfaces. The `FindInterfaces` function (shown in Listing 2-8 (page 32)) creates an iterator to iterate over all interfaces on the device and then creates a device interface to communicate with each one. For each interface found, the function opens the interface, determines how many endpoints (or pipes) it has, and prints out the properties of each pipe. Because opening

an interface causes its pipes to be instantiated, you can get access to any pipe by using its pipe index. The pipe index is the number of the pipe within the interface, ranging from one to the number of endpoints returned by `GetNumEndpoints`. You can communicate with the default control pipe (described in "USB Transfer Types" (page 11)) from any interface by using pipe index 0, but it is usually better to use the device interface functions for the device itself (see the use of `IOUSBDeviceInterface` functions in Listing 2-5 (page 28)).

The sample code employs conditional compilation using `#ifdef` and `#ifndef` to demonstrate both synchronous and asynchronous I/O. If you've chosen to test synchronous I/O, `FindInterfaces` writes the test message (defined in Listing 2-1 (page 23)) to pipe index 2 on the device and reads its echo before returning. For asynchronous I/O, `FindInterfaces` first creates an event source and adds it to the run loop created by the `main` function (shown in Listing 2-2 (page 24)). It then sets up an asynchronous write and read that will cause a notification to be sent upon completion. The completion functions `WriteCompletion` and `ReadCompletion` are shown together in Listing 2-9 (page 36).

Listing 2-8 Finding interfaces on the bulk test device

```
IOReturn FindInterfaces(IOUSBDeviceInterface **device)
{
    IOReturn                    kr;
    IOUSBFindInterfaceRequest   request;
    io_iterator_t               iterator;
    io_service_t                usbInterface;
    IOCFPlugInInterface         **plugInInterface = NULL;
    IOUSBInterfaceInterface     **interface = NULL;
    HRESULT                     result;
    SInt32                      score;
    UInt8                       interfaceClass;
    UInt8                       interfaceSubClass;
    UInt8                       interfaceNumEndpoints;
    int                         pipeRef;

#ifndef USE_ASYNC_IO
    UInt32                      numBytesRead;
    UInt32                      i;
#else
    CFRunLoopSourceRef          runLoopSource;
#endif

    //Placing the constant kIOUSBFindInterfaceDontCare into the following
    //fields of the IOUSBFindInterfaceRequest structure will allow you
    //to find all the interfaces
    request.bInterfaceClass = kIOUSBFindInterfaceDontCare;
    request.bInterfaceSubClass = kIOUSBFindInterfaceDontCare;
    request.bInterfaceProtocol = kIOUSBFindInterfaceDontCare;
    request.bAlternateSetting = kIOUSBFindInterfaceDontCare;

    //Get an iterator for the interfaces on the device
    kr = (*device)->CreateInterfaceIterator(device,
                                        &request, &iterator);
    while (usbInterface = IOIteratorNext(iterator))
    {
        //Create an intermediate plug-in
        kr = IOCreatePlugInInterfaceForService(usbInterface,
                            kIOUSBInterfaceUserClientTypeID,
                            kIOCFPlugInInterfaceID,
                            &plugInInterface, &score);
```

```
//Release the usbInterface object after getting the plug-in
kr = IOObjectRelease(usbInterface);
if ((kr != kIOReturnSuccess) || !plugInInterface)
{
    printf("Unable to create a plug-in (%08x)\n", kr);
    break;
}

//Now create the device interface for the interface
result = (*plugInInterface)->QueryInterface(plugInInterface,
            CFUUIDGetUUIDBytes(kIOUSBInterfaceInterfaceID),
            (LPVOID *) &interface);
//No longer need the intermediate plug-in
(*plugInInterface)->Release(plugInInterface);

if (result || !interface)
{
    printf("Couldn't create a device interface for the interface
            (%08x)\n", (int) result);
    break;
}

//Get interface class and subclass
kr = (*interface)->GetInterfaceClass(interface,
                                            &interfaceClass);
kr = (*interface)->GetInterfaceSubClass(interface,
                                        &interfaceSubClass);

printf("Interface class %d, subclass %d\n", interfaceClass,
                                            interfaceSubClass);

//Now open the interface. This will cause the pipes associated with
//the endpoints in the interface descriptor to be instantiated
kr = (*interface)->USBInterfaceOpen(interface);
if (kr != kIOReturnSuccess)
{
    printf("Unable to open interface (%08x)\n", kr);
    (void) (*interface)->Release(interface);
    break;
}

//Get the number of endpoints associated with this interface
kr = (*interface)->GetNumEndpoints(interface,
                                &interfaceNumEndpoints);
if (kr != kIOReturnSuccess)
{
    printf("Unable to get number of endpoints (%08x)\n", kr);
    (void) (*interface)->USBInterfaceClose(interface);
    (void) (*interface)->Release(interface);
    break;
}

printf("Interface has %d endpoints\n", interfaceNumEndpoints);
//Access each pipe in turn, starting with the pipe at index 1
//The pipe at index 0 is the default control pipe and should be
//accessed using (*usbDevice)->DeviceRequest() instead
for (pipeRef = 1; pipeRef <= interfaceNumEndpoints; pipeRef++)
{
```

```
IOReturn        kr2;
UInt8           direction;
UInt8           number;
UInt8           transferType;
UInt16          maxPacketSize;
UInt8           interval;
char            *message;

kr2 = (*interface)->GetPipeProperties(interface,
                            pipeRef, &direction,
                            &number, &transferType,
                            &maxPacketSize, &interval);
if (kr2 != kIOReturnSuccess)
    printf("Unable to get properties of pipe %d (%08x)\n",
                            pipeRef, kr2);
else
{
    printf("PipeRef %d: ", pipeRef);
    switch (direction)
    {
        case kUSBOut:
            message = "out";
            break;
        case kUSBIn:
            message = "in";
            break;
        case kUSBNone:
            message = "none";
            break;
        case kUSBAnyDirn:
            message = "any";
            break;
        default:
            message = "???";
    }
    printf("direction %s, ", message);

    switch (transferType)
    {
        case kUSBControl:
            message = "control";
            break;
        case kUSBIsoc:
            message = "isoc";
            break;
        case kUSBBulk:
            message = "bulk";
            break;
        case kUSBInterrupt:
            message = "interrupt";
            break;
        case kUSBAnyType:
            message = "any";
            break;
        default:
            message = "???";
    }
    printf("transfer type %s, maxPacketSize %d\n", message,
```

```
                                                    maxPacketSize);
            }
        }

#ifndef USE_ASYNC_IO    //Demonstrate synchronous I/O
        kr = (*interface)->WritePipe(interface, 2, kTestMessage,
                                            strlen(kTestMessage));
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to perform bulk write (%08x)\n", kr);
            (void) (*interface)->USBInterfaceClose(interface);
            (void) (*interface)->Release(interface);
            break;
        }

        printf("Wrote \"%s\" (%ld bytes) to bulk endpoint\n", kTestMessage,
                                        (UInt32) strlen(kTestMessage));

        numBytesRead = sizeof(gBuffer) - 1; //leave one byte at the end
                                             //for NULL termination
        kr = (*interface)->ReadPipe(interface, 9, gBuffer,
                                            &numBytesRead);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to perform bulk read (%08x)\n", kr);
            (void) (*interface)->USBInterfaceClose(interface);
            (void) (*interface)->Release(interface);
            break;
        }

        //Because the downloaded firmware echoes the one's complement of the
        //message, now complement the buffer contents to get the original data
        for (i = 0; i < numBytesRead; i++)
            gBuffer[i] = ~gBuffer[i];

        printf("Read \"%s\" (%ld bytes) from bulk endpoint\n", gBuffer,
                    numBytesRead);

#else   //Demonstrate asynchronous I/O
        //As with service matching notifications, to receive asynchronous
        //I/O completion notifications, you must create an event source and
        //add it to the run loop
        kr = (*interface)->CreateInterfaceAsyncEventSource(
                                    interface, &runLoopSource);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to create asynchronous event source
                                    (%08x)\n", kr);
            (void) (*interface)->USBInterfaceClose(interface);
            (void) (*interface)->Release(interface);
            break;
        }
        CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource,
                            kCFRunLoopDefaultMode);
        printf("Asynchronous event source added to run loop\n");
        bzero(gBuffer, sizeof(gBuffer));
        strcpy(gBuffer, kTestMessage);
        kr = (*interface)->WritePipeAsync(interface, 2, gBuffer,
```

```
                                    strlen(gBuffer),
                                    WriteCompletion, (void *) interface);
        if (kr != kIOReturnSuccess)
        {
            printf("Unable to perform asynchronous bulk write (%08x)\n",
                                                        kr);
            (void) (*interface)->USBInterfaceClose(interface);
            (void) (*interface)->Release(interface);
            break;
        }
#endif
        //For this test, just use first interface, so exit loop
        break;
    }
    return kr;
}
```

When an asynchronous write action is complete, the `WriteCompletion` function is called by the notification. `WriteCompletion` then calls the interface function `ReadPipeAsync` to perform an asynchronous read from the pipe. When the read is complete, control passes to `ReadCompletion` which simply prints status messages and adds a `NULL` termination to the global buffer containing the test message read from the device. The `WriteCompletion` and `ReadCompletion` functions are shown together in Listing 2-9 (page 36).

Listing 2-9 Two asynchronous I/O completion functions

```
void WriteCompletion(void *refCon, IOReturn result, void *arg0)
{
    IOUSBInterfaceInterface **interface = (IOUSBInterfaceInterface **) refCon;
    UInt32                  numBytesWritten = (UInt32) arg0;
    UInt32                  numBytesRead;

    printf("Asynchronous write complete\n");
    if (result != kIOReturnSuccess)
    {
        printf("error from asynchronous bulk write (%08x)\n", result);
        (void) (*interface)->USBInterfaceClose(interface);
        (void) (*interface)->Release(interface);
        return;
    }
    printf("Wrote \"%s\" (%ld bytes) to bulk endpoint\n", kTestMessage,
                                        numBytesWritten);

    numBytesRead = sizeof(gBuffer) - 1; //leave one byte at the end for
                                        //NULL termination
    result = (*interface)->ReadPipeAsync(interface, 9, gBuffer,
                                    numBytesRead, ReadCompletion, refCon);
    if (result != kIOReturnSuccess)
    {
        printf("Unable to perform asynchronous bulk read (%08x)\n", result);
        (void) (*interface)->USBInterfaceClose(interface);
        (void) (*interface)->Release(interface);
        return;
    }
}


void ReadCompletion(void *refCon, IOReturn result, void *arg0)
{
```

```
    IOUSBInterfaceInterface **interface = (IOUSBInterfaceInterface **) refCon;
    UInt32      numBytesRead = (UInt32) arg0;
    UInt32      i;

    printf("Asynchronous bulk read complete\n");
    if (result != kIOReturnSuccess) {
        printf("error from async bulk read (%08x)\n", result);
        (void) (*interface)->USBInterfaceClose(interface);
        (void) (*interface)->Release(interface);
        return;
    }
    //Check the complement of the buffer's contents for original data
    for (i = 0; i < numBytesRead; i++)
        gBuffer[i] = ~gBuffer[i];

    printf("Read \"%s\" (%ld bytes) from bulk endpoint\n", gBuffer,
                                                    numBytesRead);
}
```

Document Revision History

This table describes the changes to *USB Device Interface Guide*.

Date	Notes
2007-09-04	Made minor corrections.
2007-02-08	Described how to determine which version of an interface object to use when accessing a USB device or interface.
2006-04-04	Made minor corrections.
2006-03-08	Emphasized which type of device interface to get for USB devices and interfaces and clarified definition of composite class device.
2005-11-09	Made minor corrections.
2005-09-08	Added information about creating a universal binary for an application that accesses a USB device.
2005-08-11	Made minor bug fixes.
2005-06-04	Added information about low latency isochronous transactions and functions.
2005-04-29	Included discussion of USB 2.0 and associated changes to isochronous functions. Changed title from "Working With USB Device Interfaces."
2004-05-27	Fixed URL for USB Common Class Specification.
2002-11-15	First version.

Document Revision History