Driver Developer Kit

Mac OS USB DDK API Reference

Preliminary Working Draft, Revision 21

10/26/98
Technical Publications
© Apple Computer, Inc. 1998

 Apple Computer, Inc.
© 1998 Apple Computer, Inc.
All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except in the normal use of the software or to make a backup copy of the software or documentation. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Printed in the United States of America.

The Apple logo is a trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

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

Apple, the Apple logo, LaserWriter, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Intel is a registered trademark of Intel Corporation.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS PROVIDED "AS IS," AND YOU, THE DEVELOPER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Figures and Tables

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

About This Note

This document provides a high-level introduction to the features of the Universal Serial Bus (USB). It also describes the Apple Macintosh software components and programming interfaces that support USB device hardware.

This document is intended for experienced hardware and software developers interested in creating USB device drivers for the Macintosh platform. Hardware and software developers reading this document should be familiar with the information in the PCI Driver Development Kit, available on the Developer CD Series, and have a copy of the current *Universal Serial Bus Specification*, which can be found at <http://www.usb.org/developers>.

If you are not familiar with the terminology used to describe the elements that make up the USB architecture, see Appendix C, “USB Terminology,” page 127 before moving on to the rest of the material in this document.

If you are interested in finding about about the features USB provides and want to get a basic desription of the elements that make up the USB topology, you should read the introductory material in Chapter 1, “Overview,” and Chapter 2, “USB Topology and Communication.” .

If you already understand the features and topology of the USB architecture and want to get to work developing a Mac OS compatible device driver for your USB device, see the material in Chapter 3, “USB Software Components,” and the reference material in Chapter 4, “USB Services Library Reference,” and Chapter 5, “USB Manager Reference.” In addition, look at the example code provided in the Mac OS USB Device Driver Kit.

Every attempt is made at keeping the information in this document concurrent with the latest revision of the Mac OS USB software. Major differences between versions of the software are noted in Appendix A, “Changes In Mac OS USB Version 1.1,” page 117. In particular, this draft includes information related to changes between version 1.0/1.0.1 and version 1.1 of the Mac OS USB software.

IMPORTANT

The information in this note is subject to change; no representation or guarantee is made about its accuracy or completeness. ▲

Contents of This Note

The information is arranged in four chapters and three Appendices:

- Chapter 1, “Overview,” provides an introduction to the USB architecture.
- Chapter 2, “USB Topology and Communication,” provides a high level overview of the topology of the USB and how the host software communicates with devices over the USB.
- Chapter 3, “USB Software Components,” is an overview of the components that make up the Macintosh USB software architecture.
- Chapter 4, “USB Services Library Reference,” describes the Mac OS USB system software libraries that developers use to support programming USB class drivers for their USB devices.
- Chapter 5, “USB Manager Reference,” describes the Mac OS USB Manager library that provides service to the Mac OS and extension clients.
- Appendix A, “Changes In Mac OS USB Version 1.1,” provides information about the differences that developers need to be aware of in version 1.1 of the Mac OS USB software.
- Appendix B, “Conventions and Abbreviations,” provides a list of standard abbreviations used in Apple technical documentation.
- Appendix C, “USB Terminology,” defines many of the terms that are commonly used in discussion related to the USB hardware and software architecture.

Supplemental Reference Documents

For technical documentation describing the USB specification, see the Universal Serial Bus Specification, which can be found at

http://www.usb.org/developers

Technical specifications for USB device classes can also be found at the USB web site.

For information about PCI expansion cards, refer to *Designing PCI Cards and Drivers for Power Macintosh Computers.*

Developers should also have copies of the relevant books of the *Inside Macintosh* series, available in technical bookstores and on the World Wide Web at

http://developer.apple.com/techpubs/mac/

Mac OS USB Resources

For late-breaking information, technical notes, and sample code for developing USB device drivers for the Macintosh platform, visit the Mac OS USB web site:

http://developer.apple.com/dev/usb/

For developers getting started with the Macintosh platform, see the Introduction to Macintosh Programming web site at:

http://developer.apple.com/macos/intro.html

For developers creating Macintosh software for gaming devices, see the Macintosh Game Sprockets web site where you will find information about Input Sprockets version 1.3, which supports writing Input Sprockets for USB gaming devices.

http://developer.apple.com/dev/games/

Apple Developer Connection Web Site

The Apple Developer Connection Web site is the one-stop source for finding technical and marketing information specifically for developing successful Macintosh-compatible software and hardware products. Developer Connection is dedicated to providing developers with up-to-date Apple documentation for existing and emerging Macintosh technologies. Developer World can be reached at

<http://developer.apple.com/programs/>

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

C H A P T E R   1

Overview

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

This chapter provides a high-level introduction to the features of the Universal Serial Bus™ (USB).

Introduction to USB

This section describes the benefits of incorporating USB into the Macintosh hardware architecture. It also provides information about the selection of devices supported by the USB architecture.

Why Incorporate USB Into the Macintosh Architecture?

The motivation behind the selection of USB for the Macintosh architecture is simple.

- USB is a low-cost, medium-speed peripheral expansion architecture that provides data transfer rates up to 12 Mbps.
- The USB is a synchronous protocol that supports isochronous and asynchronous data and messaging transfers.
- USB provides considerably faster data throughput for devices than does the Apple Desktop Bus (ADB) and the Macintosh modem and printer ports. This makes USB an excellent replacement solution for not only the existing slower RS-422 serial channels in the Macintosh today, but also the Apple Desktop Bus, and in some cases slower speed SCSI devices.

In addition to the obvious performance advantages, USB devices are hot pluggable and as such provide a true plug and play experience for computer users. USB devices can be plugged into and unplugged from the USB anytime without having to restart the system. The appropriate USB device drivers are dynamically loaded and unloaded as necessary by the Macintosh USB system software to support hot plugging and unplugging.

Better Device Expansion Model

The USB specification includes support for up to 127 simultaneously available devices on a single computer system. (One device ID is taken by the root hub.) To connect and use USB devices, it isn't necessary to open up the system and add additional expansion cards. Device expansion is accomplished with the

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

addition of external USB multiport hubs. Hubs can also embedded in USB devices like keyboards and monitors, which provide device expansion in much the same way that the Apple Desktop Bus (ADB) is extended for the addition of a mouse through the keyboard or monitor. However, the USB implementation doesn't have the device expansion or speed limitations that ADB does.

Compact Connectors and Cables

USB devices utilize a compact 4-pin connector rather than the larger 8- to 25-pin connectors typically found on RS-232 and RS-422 serial devices. This results in smaller cables with less bulk. The compact USB connector provides two pins for power and two for data I/O. Power on the cable relieves hardware manufacturers of low-power USB devices from having to develop both a peripheral device and an external power supply, thereby reducing the cost of USB peripheral devices for manufacturers and consumers.

The cables for high-speed and low-speed devices differ in construction. High-speed USB device cables require shielding and two pairs of twisted-pair wires inside. One twisted pair provides power, nominally +5V (4.3 to 5.3 V at 100ma) for devices connected directly to the host, and ground. A powered hub can provide up to 500ma of +5V per port. (See "USB Hub Devices" (page 1-19) for a description of the services a hub provides on the USB.) The other pair of wires is for data I/O signals. (Low-speed cables are untwisted and do not require shielding.)

High-speed cables are most common, and appear as patch cables to attach hubs to hubs, or attach high-speed devices to a hub. Low-speed cable length can be up to 3 meters, and high-speed cable length up to 5 meters. Both high-speed and low-speed cables can be used on the same system bus.

USB cables are directional, the upstream connector is mechanically different from the downstream connector. The upstream connector has a small nearly square shape with a stacked pinout and the downstream connector has rectangular shape with an in-line pinout. This prevents users from connecting cables in a way that would create a loopback connection at a hub.

Use of Standard Hardware

Devices that are designed in accordance with the USB standard should not require any modification to run on a Macintosh computer or other hardware platforms. The only changes that developers need be concerned with to support

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

the Macintosh market are the changes involved in the development of Macintosh USB device drivers and applications.

Lower Cost Than Comparable Non-USB Peripherals

Low-power USB devices are less expensive than their serial or parallel interface counterparts, because of the elimination of the power supply and because the USB standard is also incorporated into PC systems developed around the PC '98 hardware architecture. Future versions of the PC '98 compliant operating systems will also include built-in driver support for a wide variety of USB devices. Together these factors mean that a larger customer base will form for USB peripheral devices, resulting in lower retail costs of USB devices for all personal computer users.

Wide Selection of USB Devices

The USB specification supports lower-speed devices, such as a keyboards, mice, joysticks, and gamepads, at 1.5 Megabits per second and higher speed devices, such as removable storage devices, scanners, or digital cameras, at up to 12 Megabits per second (high-speed is referred to as *full speed signalling* in the USB specification).

Device Classes

USB devices are categorized by class. Table 1-1 lists a few examples of USB device classes.

Table 1-1 Examples of USB device classes

USB device class	**USB devices in class**
Audio class	Speakers, microphones
Communication class	Modem, speakerphone, internet phone
Composite class	A single device that supports multiple functions, mice, keyboards, and others
HID class	Keyboards, mice, joysticks, drawing tablets, and other pointing devices

Table 1-1 Examples of USB device classes (continued)

USB device class	**USB devices in class**
Hub class	Hubs provide additional attachment points for extending the USB
Mass storage class	Floppy drives, other removable storage devices.
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

Low- and High-Speed Devices

Low-speed devices, which may include keyboards, mice, drawing tablets and others, are typically in a USB class called the Human Interface Device (HID) class. There is generally some cost reduction in low-speed devices because the cabling is less expensive than cabling for high-speed devices.

Low-speed devices support only short messaging and do not support bulk and isochronous transfers.

High-speed devices generally include communications devices, printing devices, bulk storage devices, audio devices, and others.

There is nothing to prevent USB devices from being in either a high-speed or low-speed category. However, some classes of devices, those that require bulk or isochronous transfer services, cannot be part of the low-speed category.

Note
High speed in the case of USB is not comparable to high-speed devices on a FireWire bus. USB is a complementary technology to FireWire, not a competing technology. USB enables the use of affordable higher-speed consumer grade peripherals on Macintosh computers.

USB Hub Devices

Hubs are also USB devices and provide attachment points to the USB for other devices or hubs. Hubs can be embedded into other USB devices (this is known as a compound class device). For example, a hub can reside in a keyboard,

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

monitor, or printer to provide attachment points for other (typically) low-power devices.

Hubs are also in the form of standalone multi-port hubs that provide attachment points to the USB for other USB devices. Multiport-hubs are generally categorized as bus-powered and self-powered. Bus-powered hubs can request a total of 500ma from the USB and provide no more than 100ma of power at each port on the hub. Even though a bus-powered hub may request 500ma, it may not get the power depending on the devices connected upstream on the USB. Self-powered hubs (hubs that include a source of power external to the USB) can supply additional power to the USB, and are required to provide up to 500ma at each port on the hub.

While it is physically possible to connect two bus-powered hubs together in-line without damaging any devices on the USB, it should not be done because there isn't enough power on the USB to support such an attachment. If sufficient power isn't available for the downstream device, the USB software will not be able to properly configure the device's power requirements. The downstream hub most likely will not function. However, a self-powered hub and bus-powered hub can be connected together in-line.

See Chapter 11 of the Universal Serial Bus Specification for additional information about USB hubs.

The USB Root Hub

There is also a hub referred to as the root hub. The root hub is a software simulation of a hub with hardware controller support. It acts as part of the host hardware environment on the main logic board or on an I/O expansion card. The root hub is similar to the other hubs, in that it provides an attachment point or points to extend the USB from the host, however it is the initial connection point and parent of the bus at which all signals originate. A simple diagram of the USB topology is shown in Figure 2-1 (page 2-27).

Compatibility Issues

This section describes issues related to compatibility with legacy Macintosh ADB, serial/LocalTalk, and storage devices. In addition, it describes some fundamental differences in how USB works as a serial communications channel in the Macintosh environment.

USB Software Gestalt Selectors

There are four gestalt selectors defined for determining the version attributes of the USB software. To use the gestalt selectors you must understand how to use the Gestalt Manager, which is defined in *Inside Macintosh: Overview*. The gestalt selectors for USB software are defined in Chapter 3, "USB Software Presence and Version Attributes."

ADB, Serial/LocalTalk, and USB

You cannot physically connect legacy ADB devices or serial/LocalTalk devices to USB ports.

It is currently not possible to use a USB keyboard to access OpenFirmware if the keyboard is connected to a PCI USB controller card in a Macintosh. Essentially, keystrokes are not recognized early enough in the boot sequence to allow boot keyboard access to OpenFimware. Other keyboard key combinations, such as turning off system extensions with the Shift key down, do function as expected.

Macintosh-To-Macintosh USB Connections

USB is a serial communications channel, but does not replace LocalTalk functionality on Macintosh computers. You cannot connect two Macintosh computers together using the USB like you can in a LocalTalk serial network for a couple of reasons.

- The USB cable connectors are designed in such a way that it should be impossible to attach two upstream devices together. A standard USB cable has one upstream connector and one downstream connector. The root hub in the Macintosh computer is the first device on the USB, and as such it is always an upstream device in the USB topology.
- The USB uses a master/slave communication model in which the Macintosh host controls all communication and is the master of the bus. There cannot be two masters on the same bus.

The most cost efficient method for networking USB enabled Macintosh computers together is through the built-in Ethernet port.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

USB Storage Devices

Version 1.0 of the Apple USB software does not support booting from any USB storage device.

Keyboard Requirements

Apple provides a HID class driver for the Apple USB keyboard, which supports the USB boot protocol. Keyboards intended for use on the Macintosh platform must support the HID boot protocol, as defined in the USB Device Class Definition for Human Interface Devices (HIDs).

USB Data Transfer Types Supported

There are four data transfer types defined by the USB specification. They are

- Bulk transfers which offer guaranteed delivery of data. This may include retrying transmissions at the hardware level. Bulk data transactions are best suited for printers, scanners, modems, and devices that require accurate delivery of data with relaxed timing constraints.
- Interrupt transfers, which allow a device to signal the host. Interrupt data transactions do not use up CPU cycles unless the device has data ready. Interrupt transactions are used for HID class devices like keyboards, mice, joysticks, as well as devices that want to report status changes, such as serial or parallel adaptors and modems.
- Isochronous transfers for one time delivery of data. Isochronous data transactions are best suited for audio or video data streams.
- Control transfers for device configuration and initialization.

Version 1.0 of the Mac OS USB software provides functions that support only control, bulk, and interrupt transfer types. Version 1.1 supports control, bulk, interrupt, and isochronous transfers types.

USB Controller Support

The Apple Macintosh USB system software supports controllers compatible with the Open Host Controller Interface (OHCI) specification. It does not support Universal Host Controller Interface (UHCI) controllers.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Some early USB devices (most notably keyboards) can't interoperate with an OHCI controller. These devices will not be supported by the Apple USB system software.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

C H A P T E R 2

USB Topology and Communication

This chapter introduces the topology of the USB and how the host software communicates with devices over the USB. This is only a high-level introduction. For the complete details of the USB topology and communication model, see the Universal Serial Bus Specification, which can be found at <http://www.usb.org/developers>.

USB Bus Topology

This section briefly describes the topology and communication model for the USB architecture.

The USB architecture has a well defined physical and logical bus topology, which is fully described in the Universal Serial Bus Specification. The physical topology defines how USB devices are connected together. The logical topology defines how the various components that make up the physical topology are viewed by the host software.

Host Software

The client software, the USB management software, and the USB host controller together make up the host software in the USB logical topology.

The host plays a special role as the arbiter of all activity on the USB. A USB device can only gain access to the bus through the host by supplying a device descriptor that includes the information necessary to manage the device according to its features and class identifiers. See Chapter 4, "USB Services Library Reference," for additional information about the contents of the device configuration descriptor structure.

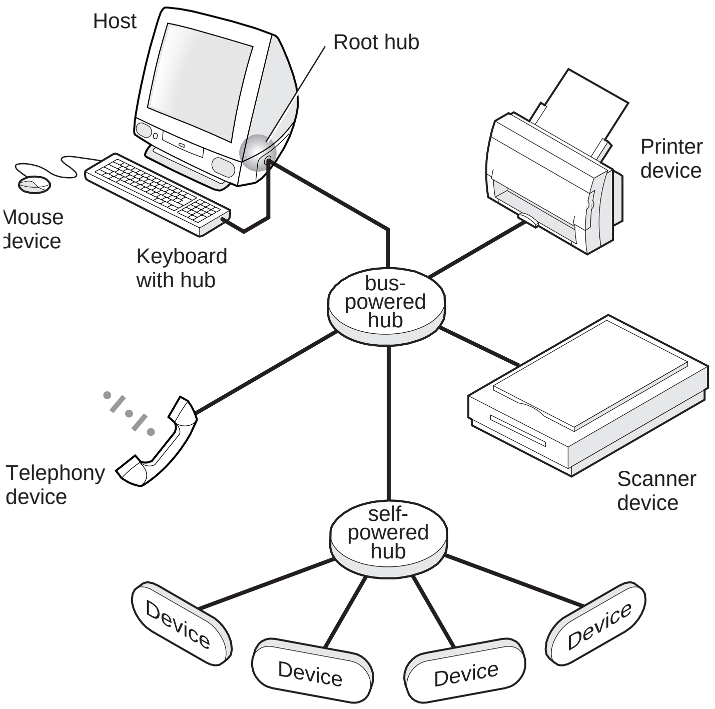
The host interacts with USB devices through the host controller. The host is responsible for:

- Monitoring the attachment and removal of USB devices
- Managing control and data flow between the host and USB devices
- Maintaining device status and activity information
- Providing a limited amount of power to the USB

Physical Topology

An example of the USB physical topology is shown in Figure 2-1. The system software has to know about the physical topology to perform bandwidth measurements in order to optimize bit time requirements for the USB as it grows with additional hubs and devices. Device drivers do not have to know about the physical topology. The USB specification states that the host can handle up to six levels of hub support.

Figure 2-1 USB physical topology



Logical Topology

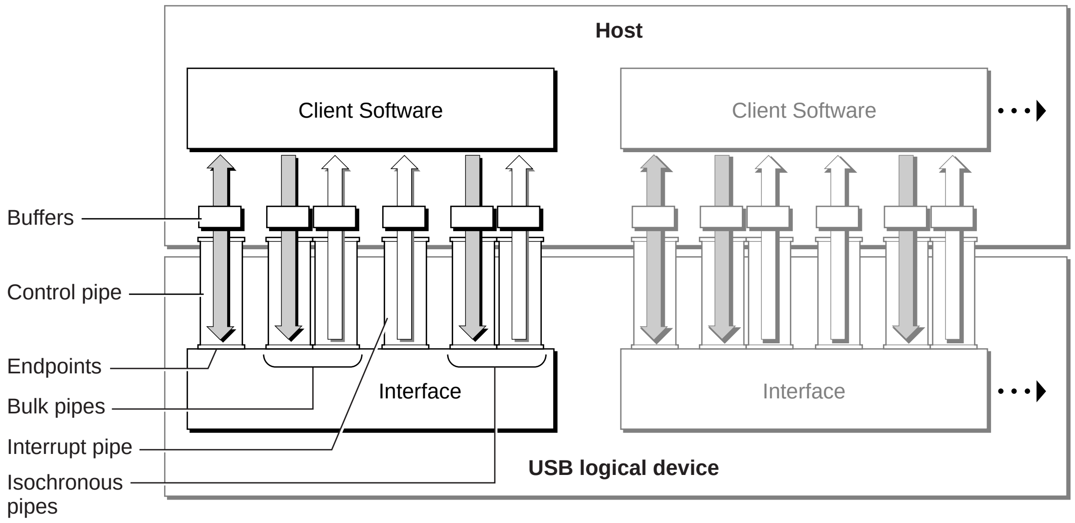
The logical topology is how the host software views and communicates with devices in the physical topology. From the host software perspective, the USB is seen as a linear addressing space.The host is aware of the physical topology so that it can accurately support connection and disconnection on hubs with attached devices.

Communication Over the USB

This section provides an abridged description of the logical communication model on USB. Refer to the Universal Serial Bus Specification for complete technical details. A simplified diagram of the USB communication flow is shown in Figure 2-2.

The USB driver software maintains an abstract view of the logical and physical topology of the bus when it communicates with USB devices. Drivers look for the interface(s) of interest that are available in devices on the USB.

Figure 2-2 USB communication flow



USB Interface

Interfaces are a means of determining the functionality a device can provide to the host and the means by which the device is controlled. For example, a device which provides a bulk interface function to the host would be controlled by an driver that understands the interface for the bulk transaction protocol.

Physical devices may contain multiple interfaces, which logically appear as device functions within devices. Each device has one interface for each function is supports.

The logical device is identified through an interface. Drivers use the USB Manager APIs to open interfaces to device functions (capabilities). The device's function(s) are defined by the interface class, subclass, and protocol values in the interface descriptor for the device.

A logical USB device is a collection of endpoints, grouped into endpoint sets, which implement a logical interface. USB software manages the interface using a pipe or pipe bundles. (Pipe bundles are used for bulk and isochronous transfers.) Data is packetized in a USB-defined structure by the host controller and moved across the USB between a software serial interface engine on the host and an endpoint on the device.

USB Devices

Every USB devices is accessed by a unique USB address, which is assigned by the USB host software after initial device recognition and configuration takes place. Each USB device additionally supports one or more endpoints with which the host may communicate. All USB devices must support a specially designated Endpoint 0 to which the USB device's default control pipe is attached during device initialization.

Endpoints

Endpoints are the terminus of a communication flow between a USB device and the host. Endpoints are a logical point inside the USB device to which the host may attach a pipe to initiate communication with a USB device. Endpoints represent a specific data connection where interfaces represent a larger functional connection.

Endpoint 0

Endpoint 0 has a special responsibility. It is used for USB device initialization and configuration. All USB devices must support a default endpoint 0. Endpoint 0 supports control transfers which provide control pipe access to device descriptors and control requests to modify the device's behavior.

Non-0 Endpoints

Non-0 endpoints are endpoints greater than 0. Low speed functions are limited to two optional endpoints beyond the required endpoint 0. Higher speed devices can have additional endpoints. However, no more than 16 input endpoints and 16 output endpoints. Endpoint 0 is used as both an input and output endpoint, which leaves a total of 15 each for input and output endpoints (0 through 15 = 16).

A non-0 endpoint is not available for use until it is configured by the startup configuration process when the device is attached to the USB.

Non-0 endpoints are not unique across device configurations. Endpoint numbers are defined by the device vendor in a configuration descriptor for the device. The associated interface or function associated with an endpoint number may be different for the same endpoint number in different devices. You should not count on endpoint numbers being identical from device to device for a given interface.

Pipes

A pipe represents the communication link between a USB device endpoint and the host software. Data moves to and from the USB device through the pipe. Pipes have two communication modes, stream and message, and four transfer types, control, isochronous, interrupt, and bulk.

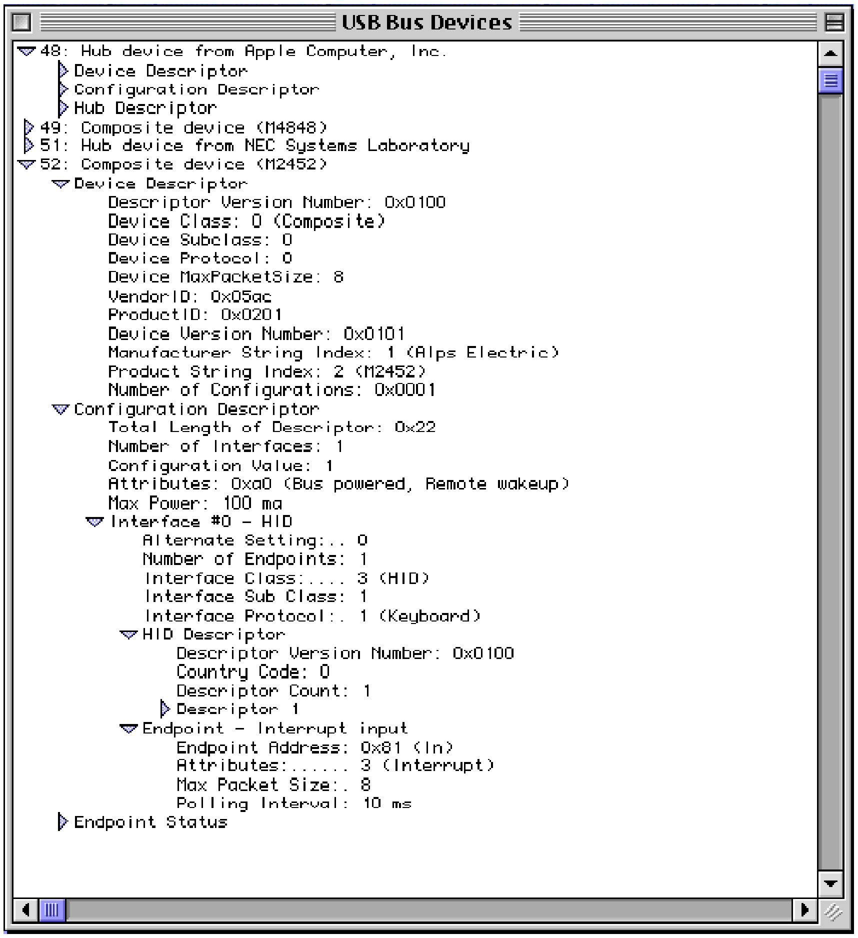
For a detailed descriptions of USB interfaces, endpoints, pipes, communication modes, and transfer types see the Universal Serial Bus Specification, which can be found at <http://www.usb.org>.

A Look At USB Devices with USB Prober

The USB Prober application, which is part of the Mac OS USB Device Driver Kit, is a utility for examining devices on the USB. Figure 2-3 is a screen capture taken from the USB Prober application. The example shows the various descriptors that define a USB keyboard class device. You can see that the keyboard is a composite class device, that has a HID interface with a keyboard protocol, and has a single endpoint that supports the interrupt transfer type.

Additional details about how to access the information that defines a USB device can be found in “USB Services Library Reference.”

Figure 2-3 USB Prober view of a USB device



```
▽48: Hub device from Apple Computer, Inc.
    ▷Device Descriptor
    ▷Configuration Descriptor
    ▷Hub Descriptor
▷49: Composite device (M4848)
▷51: Hub device from NEC Systems Laboratory
▽52: Composite device (M2452)
    ▽Device Descriptor
        Descriptor Version Number: 0x0100
        Device Class: 0 (Composite)
        Device Subclass: 0
        Device Protocol: 0
        Device MaxPacketSize: 8
        VendorID: 0x05ac
        ProductID: 0x0201
        Device Version Number: 0x0101
        Manufacturer String Index: 1 (Alps Electric)
        Product String Index: 2 (M2452)
        Number of Configurations: 0x0001
    ▽Configuration Descriptor
        Total Length of Descriptor: 0x22
        Number of Interfaces: 1
        Configuration Value: 1
        Attributes: 0xa0 (Bus powered, Remote wakeup)
        Max Power: 100 ma
      ▽Interface #0 - HID
            Alternate Setting:.. 0
            Number of Endpoints: 1
            Interface Class:.... 3 (HID)
            Interface Sub Class: 1
            Interface Protocol:. 1 (Keyboard)
          ▽HID Descriptor
                Descriptor Version Number: 0x0100
                Country Code: 0
                Descriptor Count: 1
              ▷Descriptor 1
          ▽Endpoint - Interrupt input
                Endpoint Address: 0x81 (In)
                Attributes:...... 3 (Interrupt)
                Max Packet Size:. 8
                Polling Interval: 10 ms
    ▷Endpoint Status
```

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

C H A P T E R 3

USB Software Components

This chapter is in preliminary overview of the components that make up the Macintosh USB software architecture.

Mac OS Software for USB Devices

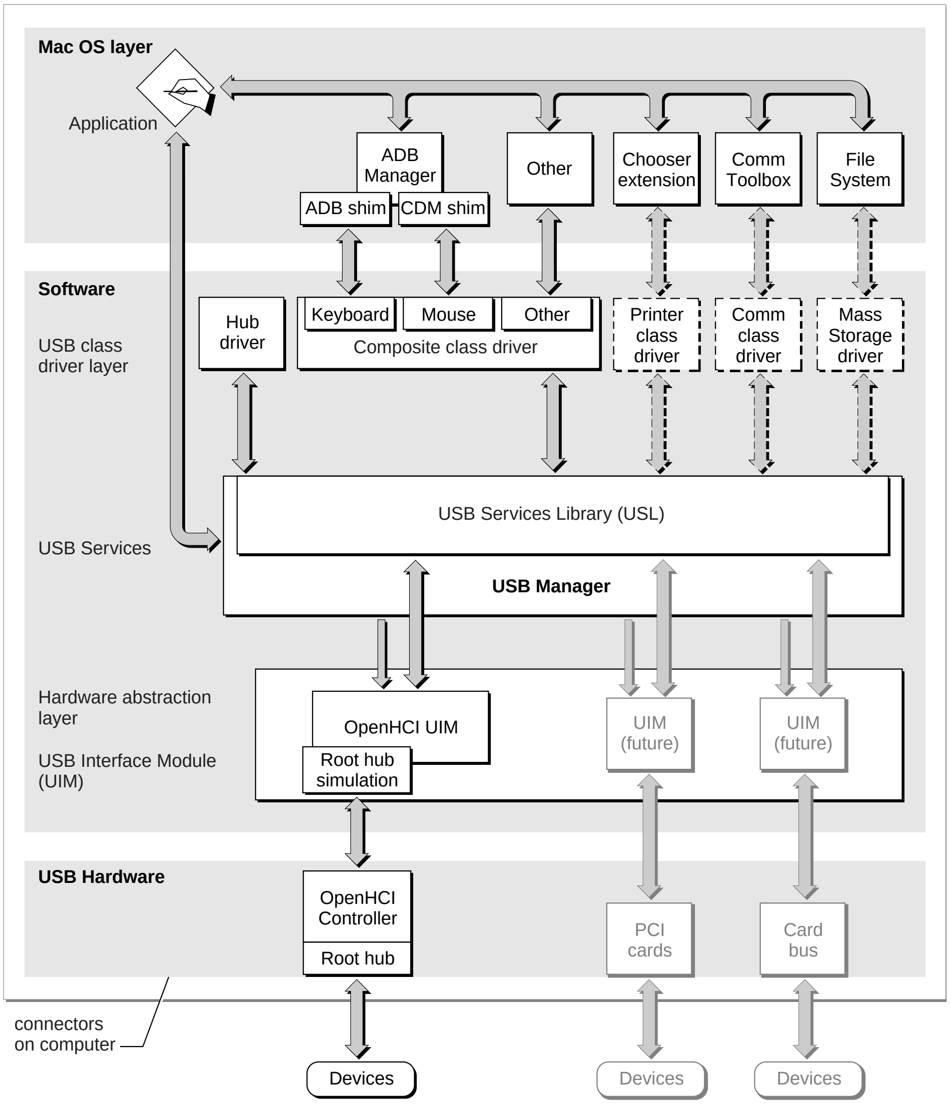
The software that supports USB devices in the Mac OS environment includes a USB Interface Module (UIM), a USB Manager, a USB Services Library (USL), and USB class drivers.

- The UIM, pronounced *whim*, communicates with the USB controller hardware and provides a hardware abstraction layer for the USL and USB Manager.
- The USB Manager is the API provided to the Mac OS, or extensions that need information related to the USB.
- The USL is the API that USB device drivers use to add device functionality to the USB on Macintosh computers. The API is defined in the USB.h file and Chapter 4, "USB Services Library Reference."
- Device class drivers in version 1. 0 of the Macintosh USB system software include a USB composite device class driver to support USB HIDs, such as keyboards and mice, and a hub class driver to support hubs attached to the USB.

The Mac OS USB system software components are available as system extensions for Macintosh systems that do not include built-in USB ports. This generally applies to Macintosh computers that have USB ports on PCI cards. Macintosh computers that have USB ports designed into the main logic board, like the iMac computer, have all of the USB software components, excluding the class drivers, in the Macintosh system ROM.

Figure 3-1 shows the components that make up the USB software architecture on the Macintosh computer. Release version 1.0 of the Macintosh USB software provides only class driver support for a USB keyboard, mouse, and hub.

Figure 3-1 USB architecture



USB Software Presence and Version Attributes

Applications can obtain information about the presence, version, and attributes of USB software on Macintosh computers by using the Gestalt Manager routines and USB gestalt selectors. The Gestalt Manager is defined in *Inside Macintosh: Overview*.

The gestaltUSBAttr, gestaltUSBPresent, gestaltUSBHasIsoch, and gestaltUSBVersionGestalt selectors are defined for Macintosh USB software as follows:

```
gestaltUSBAttr      = ‘usb ‘    USB attributes
gestaltUSBPresent   = 0         Bit 0 is set if USB software is present
gestaltUSBHasIsoch  = 1         Bit 1 is set is USB software supports
                                isochronous transfers
gestaltUSBVersion   = ‘usbv‘    USB version number
```

The gestaltUSBVersion selector returns the version of the USB software in a 32-bit format as follows:

```
MMmmRRss
```

MM	The most significant byte containing the major version number. The current value for the major version number is 1. This number will increment with each major release.
mm	The next byte contains the minor version number. The current value for the minor version number is 1.
RR	The next byte contains the release stage. The release stage is defined as: 0x02 = development, 0x04 = alpha, 0x06 = beta, and 0x08 = final. If the software was at the beta release stage, this number would be 0x06.
ss	The least significant byte is the sequence number of the release, and it changes with every build of the USB software.

USB Interface Module (UIM)

The UIM provides the upper layers of the USB software with a hardware abstraction layer to the USB host controller interface hardware. The UIM communicates directly with the USB controller hardware to set up the appropriate communication links with the USB devices on the bus. The UIM also provides root hub simulation.

The UIM is a native driver 'ndrv' code fragment, as defined in “Designing Macintosh PCI Cards and Drivers.” In addition to supporting the data export guidelines for ndrvs, the UIM provides USB specific data exports that define the UIM driver entry points and descriptor structures built for devices on the USB.

OpenFirmware builds a Name Registry entry for the host controller and matching UIM during hardware bring up time, prior to booting the operating system. The USB Manager uses the information in the Name Registry to communicate with the UIM. Once the UIM is loaded and begins hub simulation, the USB Manager determines that a hub is present and loads the hub driver. At this point, the hub driver begins monitoring all USB device activity at the USB hub simulation provided by the UIM.

A UIM is required for every USB bus controller implementation installed in the host. For example, multiple UIMs would be required on a host which has both a built-in USB host controller interface and a USB controller interface on a PCI card. Developers designing PCI, Card Bus, or any other controller interface to the USB may need to provide a UIM for their card interface. For information regarding the APIs needed for UIM software development, send email to the Apple USB evangelist at USB@apple.com.

USB Manager

The USB Manager performs driver matching and loading services and communicates internally with other components of the Macintosh USB host software to identify devices on the USB. The USB Manager also provides services that Mac OS and applications use to determine the status of devices, handle power management tasks, and notify the user or other applications about USB devices being attached to or removed from the USB.

An example of a service that the USB Manager can provide for a client is when a client makes a request to find a keyboard. The USB Manager determines if a keyboard is installed and returns the appropriate response. If a keyboard is installed, the client can ask where the class driver is for that keyboard. The USB Manager then points to the code fragment that contains the class driver for keyboards. The client then communicates with the keyboard through the class driver. The keyboard class driver communicates with the keyboard interface through the USB Services Library. The API for the USB Manager is included in Chapter 5, “USB Manager Reference.”.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Hub Driver

The hub driver provides support for the USB software architecture by monitoring the connection and removal of devices on the USB at a hub. This process is referred to as device enumeration. When the hub driver recognizes that a device has been plugged into the bus at a given port ID on a hub, it notifies the USL. The USL notifies the USB Manager, which in turn builds the Name Registry entry for the device and binds the appropriate class driver with that device. When the hub driver finds a device, it notifies the USB Manager that a device has been found. The USB Manager loads the appropriate class driver based on the class and subclass and other information found in the device configuration or interface configuration descriptor for the device.

Additional information about the process of bus enumeration is described in Chapter 9 of the Universal Serial Bus Specification.

USB Class Drivers

A USB device must have a USB class driver or drivers for every interface (function) the device supports to operate properly on the Macintosh computer. Macintosh USB class drivers are “ndrv” native code fragments and as such follow the guidelines specified for creating Macintosh native drivers in the Designing Macintosh PCI Cards and Drivers. USB class drivers provide a driver descriptor structure that the USB Manager uses to match with a device or interface.

The USB Manager matches drivers to device interfaces by initially examining the product ID and vendor ID fields in the device descriptor for the device. To ensure proper device and driver matching, additional information regarding the device is examined if none or more than one driver matches the product ID and vendor ID values for the device. Detailed information about how USB class driver and device matching is accomplished is in the Universal Serial Bus Common Class Specification which can be found at <http://www.usb.org>.

There are several classes of drivers defined by various USB specifications, and new classes are being proposed all the time. The Macintosh USB software includes HID class drivers for the HID interfaces in the USB keyboard and mouse. The keyboard and mouse drivers are loaded by a composite class driver, which is loaded by the hub driver when the keyboard and mouse are found on the USB.

A USB device includes an interface or interfaces, which are defined in descriptor data structures associated with the device. The interfaces are like sub devices within the device, each having a function specified by numerical class and sub class identifiers. The functions provide device capabilities to the host system. Interfaces also define how a function in a device is accessed by the host system. The functional features of the device are accessed by the USL when given an interface reference.

The Macintosh system software maintains a driver dispatch table for USB class drivers that defines among other things the driver initialization routine, driver gestalt, and the driver callback completion routine. For more information, see the driver descriptor structure defined in the USB.h file.

USB Services Library (USL)

The USB Services library is the programming interface that USB device drivers use to communicate with the USB on a Macintosh computer. The USL provides the services necessary to find a device with the appropriate interface, open an interface to the device, open the device, instantiate the appropriate pipe connections, determine device status, and perform read and write transactions with the device. For additional information about the USL, see Chapter 4, “USB Services Library Reference.”

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

C H A P T E R 4

USB Services Library Reference

This chapter describes the APIs in the Mac OS USB Services Library that support software development of USB class drivers.

The Mac OS USB DDK provides source code for building examples of USB class drivers. The driver sources illustrate how to use the USL for USB class driver development. The USB.h file contains the current application programming interfaces to the USL. Future class driver compatibility requires adhering to the interfaces defined in the USL libraries. The Mac OS USB DDK and other valuable resources for developers can be found at:

http://developer.apple.com/dev/usb/devinfo.htm

The Mac OS USB DDK includes a folder called Writing a USB Driver. The USBDriversTechnote.html file in that folder describes how to write a USB class driver, and discusses how Mac OS processes communicate with the device via the class driver.

The Mac OS USB DDK ReadMe file in the Mac OS USB DDK folder provides the instructions for setting up your Macintosh development system and target environments.

Mac OS USB Compatibility Notes file contains information about ADB, serial port, and USB gaming device compatibility and software support issues.

USB Services Library (USL)

The USB Services Library is the programming interface that USB device drivers use to communicate with the USB on a Macintosh computer. The USL provides all of the control and status functions necessary to find a device with the appropriate interface, open an interface to the device, open the device, instantiate the appropriate pipe connections, determine device status, and perform read and write transactions with the device.

Errors And Error Reporting Conventions

The USB software uses a “return errors, set references” convention. In this convention, all APIs return a common `OSStatus` type. `PipeRef`, `DeviceRef`, `InterfaceRef`, and `endpointRef` reference numbers are all in a field of the parameter block passed to the function. No reference variables do double duty. That is, they do not report both error codes and reference numbers (refnums).

There is one exception to this however, a reference number of `nil` is an indication that the reference number has not been set properly.

Error codes returned by the USL are in the range -6900 to -6999 and are listed in "USL Error Codes" (page 4-104).

The following discussion deals with common causes of errors returned by the USL

Device Access Errors

Any function that accesses a device may give one of the transfer mode errors. Transfer mode errors cause a pipe stall on non-default pipes. The transfer errors are in the range -6901 through -6915 as follows:

`kUSBCRCErr`	`-6913`	Pipe stall, bad CRC
`kUSBBitstufErr`	`-6914`	Pipe stall, bitstuffing
`kUSBDataToggleErr`	`-6913`	Pipe stall, bad data toggle
`kUSBEndpointStallErr`	`-6912`	Device didn't understand
`kUSBNotRespondingErr`	`-6911`	Pipe stall, no device, device hung
`kUSBPIDCheckErr`	`-6910`	Pipe stall, PID CRC error
`kUSBWrongPIDErr`	`-6909`	Pipe stall, bad or wrong PID
`kUSBOverRunErr`	`-6908`	Packet too large or more data than allocated buffer
`kUSBUnderRunErr`	`-6907`	Less data than buffer
`kUSBBufOvrRunErr`	`-6904`	Host hardware failure on data in, PCI busy?
`kUSBBufUnderRunErr`	`-6903`	Host hardware failure on data out, PCI busy?
`kUSBNotSent1Err`	`-6902`	Transaction not sent
`kUSBNotSent2Err`	`-6901`	Transaction not sent

The `kUSBNotRespondingError` most often occurs when a device is unplugged. A driver should prepare to be deleted, if it gets this error. This error may occur when a device is hung, or when a bus error occurs.

Errors on the USB Bus

Errors on the USB bus occur when a device is behaving erratically or there are bad cables or connectors. USB bus errors include:

`kUSBCRCErr`	`-6913`	Pipe stall, bad CRC
`kUSBBitsufErr`	`-6914`	Pipe stall, bitstuffing
`kUSBDataToggleErr`	`-6913`	Pipe stall, bad data toggle
`kUSBPIDCheckErr`	`-6910`	Pipe stall, PID CRC error
`kUSBWrongPIDErr`	`-6909`	Pipe stall, bad or wrong PID

The USB bus errors are uncommon and should rarely be seen.

Incorrect Command Errors

When a device receives an incorrect command or a command it cannot comply with, it stalls the pipe and returns a `kUSBEndpointStallErr`.

Driver Logic Errors

The `kUSBOverRunErr` and `kUSBUnderRunErr` are usually caused by logic errors in the driver. In most cases, the driver and the device are not in agreement as to how much data is to be transferred.

An over run error occurs most often when a buffer is not an exact multiple of the maximum packet size (`maxPacketSize`), and the controller determines that the last packet will overflow the end of the buffer. This also occurs if a packet is sent that is larger than the maximum packet size. This is often a protocol error and the sign of a bad device.

In version 1.0 of the USB Services Library software, this error can occur if the transfer buffer is not aligned to the maximum packet size of the endpoint. This problem will be addressed in a later revision of the Mac OS USB software.

Under run errors occur when a packet shorter that the maximum packet size is received. It is the pipe policy to treat this situation as an error. In version 1.0 underrun errors are usually not generated. Short packets always cause a normal completion. The option to make short packets an error will be addressed in a later revision of the USB software.

PCI Bus Busy Errors

Errors may be generated if the PCI bus is busy for extended periods of time. These errors include `kUSBBufOverRunErr` and `kUSBBufUnderRunErr`.

USB References

All references to the USB are made on the basis of a USB reference. The USB references are of type `USBReference`. `USBDeviceRef`, `USBInterfaceRef`, `USBPipeRef`, and `USBEndPointRef` are USB references that you pass into or obtain from the USL functions. The USB reference is an opaque reference maintained by the USB Services Library.

A device reference is obtained when the class or interface driver is initialized, since it is passed as a parameter to the initialization procedure. The USB reference for a particular USB device can be found in the device entry in the Name Registry and vice-versa by calling the USB Manager. See "Topology Database Access Functions" (page 5-109) for a description of the functions available for obtaining information about USB devices.

The USBPB Parameter Block

The majority of calls to the USL are made with a parameter block of type USBPB. The USBPB parameter block contains all the necessary parameters to facilitate host communication with the device interface or device interface communication with the USB. The parameter block also includes a pointer to a callback completion routine for support of asynchronous calls.

There are currently two version of the `USBPB` parameter block, the version 1.0 parameter block (`kUSBCurrentPBVersion`), defined in this section, and the version 1.1 parameter block (`kUSBIsocPBVersion`), defined in Appendix A, that supports isochronous transfers. Version 1.1 of the Mac OS USB software accepts function calls made with both parameter blocks. For information about converting existing code to use the 1.1 USBPB, see "Code Changes Required To Support The Version 1.1 USBPB" (page A-123).

Parameters for the `USBPB` parameter block that are not specified as required in the USL function descriptions are ignored by the USL. Parameters that are not specified as output values are not altered, except for the `Reserved`, `usbWValue`, and `usbWIndex` fields.

The types associated with the version 1.0 `USBPB` parameter block structure and the USBPB structure are defined as follows:

```
typedef SInt32          USBReference;
typedef USBReference    USBDeviceRef;
typedef USBReference    USBInterfaceRef;
typedef USBReference    USBPipeRef;
typedef USBReference    USBBusRef;
typedef UInt32          USBPipeState;
typedef UInt32          USBCount;
typedef UInt32          USBFlags;
typedef UInt8           USBRequest;
typedef UInt8           USBDirection;
typedef UInt8           USBRecipient;
typedef UInt8           USBRqType;
typedef UInt16          USBRqIndex;
typedef UInt16          USBRqValue;

typedef void (*USBCompletion)(USBPB *pb);
```

The version 1.0 USBPB parameter block is defined as:

```
struct USBPB{

    void* qlink;
    UInt16 qType;
    UInt16 pbLength;             /* Length of parameter block */
    UInt16 pbVersion;            /* Parameter block version number */
    UInt16 reserved1;            /* Reserved */
    UInt32 reserved2;            /* Reserved */

    OSStatus usbStatus;          /* Completion status of the call */
    USBCompletion usbCompletion; /* Completion routine */
    UInt32 usbRefcon;            /* For use by the completion routine */
    USBReference usbReference;   /* Device, pipe, interface, endpoint */
                                 /* reference as appropriate */

    void* usbBuffer;             /* Pointer to the data to be sent */
                                 /* to or received from the device */
    USBCount usbReqCount;        /* Length of usbBuffer */
    USBCount usbActCount;        /* Number of bytes sent or received */
    USBFlags usbFlags;           /* Miscellaneous flags */

    UInt8 usbBMRequestType;      /* For control transactions, */
                                 /* the bmRequestType field */
    UInt8 usbBRequest;           /* Specific control request */
    USBRqValue usbWValue;        /* For control transactions, the */
                                 /* Value field of the setup packet */
    USBRqIndex usbWIndex;        /* For control transactions, the */
                                 /* Index field of the setup packet */
    UInt16 reserved4;            /* Reserved */
    UInt32 usbFrame;             /* Reserved for future use */
    UInt8 usbClassType;           /* Class for interfaces, */
                                  /* transfer type for endpoints */
    UInt8 usbSubclass;            /* Subclass for interfaces */
    UInt8 usbProtocol;            /* Protocol for interfaces */
    UInt8 usbOther;               /* General-purpose value */
    UInt32 reserved6;             /* Reserved */
    UInt16 reserved7;             /* Reserved */
    UInt16 reserved8;             /* Reserved */

    }USBPB;
```

During asynchronous calls, before the callback, no fields in the parameter block are valid other than the `usbRefcon` field. The `usbRefcon` field is never altered and is free for use by class drivers.

The `USBPB` parameter block has to be at least the minimum size. The size can be extended; the `pbLength` field should contain the extended size.

The current version of the parameter block is represented as a binary-coded decimal number. For version 1.0 it is of the form 0x010. It is subject to change at any time. Use the constant `kUSBCurrentPBVersion` to make sure you have the version of the parameter block described by the latest revision of this document. The isochronous variant of the parameter block is version 1.1 (`kUSBIsocPBVersion`). The version 1.1 parameter block is defined in "Changes In Mac OS USB Version 1.1" (page A-117).

The values passed in the `usbBMRequestType` field require a specific format, which can be derived by using the `USBMakeBMRequestType` function (page 4-62).

Required USB Parameter Block Fields

All calls to the USL that require a `USBPB` parameter block must supply the these fields:

`pbLength`	Length of the `USBPB` parameter block, including any client additions
`pbVersion`	Version number of `USBPB` in binary-coded decimal, currently 1.1, initialize to `kUSBCurrentPBVersion` or `kUSBIsocPBVersion` for isochronous call support. See "Changes In Mac OS USB Version 1.1" (page A-117) for additional information about how it use the isochronous variant of the USB parameter block.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

`usbCompletion`	Completion routine
`usbRefcon`	For client use
`usbFlags`	Unless otherwise specified in the function description, should be set to 0

The listed fields may not explicitly be referenced in all the call descriptions in this document, but they are required.

Standard Parameter Block Errors

All of the functions that use the parameter block return errors when a bad value was passed in the parameter block. The standard parameter block errors are listed in Table 4-1.

Table 4-1 Standard parameter block errors

Error constant	Error code	Definition
`kUSBPBVersionError`	-6986	The pbVersion field of the parameter block contains an incorrect version number.
`kUSBPBLengthError`	-6985	The pbLength field of the parameter block contains a value that is smaller than the sizeof(USBPB)
`kUSBCompletionError`	-6984	The usbCompletion pointer is nil or is set to kUSBNoCallBack and the function does not support this behavior
`kUSBFlagsError`	-6983	An unspecified flag has been set

Asynchronous Call Support

As a general rule, function calls to the USL complete asynchronously, with the exception of the functions listed here:

```
USBGetPipeStatusByReference
USBAbortPipeByReference
USBResetPipeByReference
USBClearPipeStallByReference
```

```
USBSetPipeIdleByReference
USBSetPipeActiveByReference
USBGetFrameNumberImmediate
USBFindNextEndpointDescriptorImmediate
USBFindNextInterfaceDescriptorImmediate
```

Since most USL functions complete asynchronously, it's important to allocate a parameter block in memory that will be available until the call completes, either with a call back or with an immediately returned error. Unless there is an immediate error, the parameter block cannot be reused until the completion routine is called. You can force the completion routine to be called for pipe transactions by calling the `USBAbortPipeByReference` function.

Asynchronous calls to the USL are supported by a completion routine mechanism. You pass a pointer to a completion routine in the `USBPB` parameter block. The completion routine is invoked when the USL function call completes, informing the driver of the calls completion.

The USB completion routine is of this form:

```
typedef void (*USBCompletion)(USBPB *pb);
```

The fields required in the `USBPB` parameter block for all USL functions that return asynchronously are

-->	`pbLength`	Length of parameter block
-->	`pbVersion`	Parameter block version number
-->	`usbCompletion`	The completion routine
-->	`usbRefcon`	General-purpose value passed back to the completion routine

During the call to the completion routine, these fields are valid:

-->	`usbStatus`	Status information
-->	`usbRefcon`	General-purpose value passed back to the completion routine
-->		Any other call-specific fields marked as output from the call
-->		Any other call-specific fields used as input to and not output from the call

When the completion routine is called, the processing of the parameter block is complete and the parameter block is again available for use. The completion

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

routine may use the same parameter block to make a new call to the USL. Polling the `usbStatus` field is not supported.

The execution level that the completion routine may be called back at is not guaranteed, unless otherwise specified in the individual routine specification. Completion usually occurs at secondary interrupt level, or at system task level. If the execution context is important to the operation of the code, the driver services call `CurrentExecutionLevel` can be used to discover the current execution level.

The driver services function `CallSecondaryInterruptHandler2` can be used to continue execution at secondary interrupt level. The `USBDelay`* function can be used to effect a transition to task level. Note system task level is not the same as application task level, it may not be safe to make some Mac OS calls, particularly file system calls at system task level. Unless otherwise specified, all of the USL functions are safe to be called from either secondary interrupt level or from system task level.

*This functionality is missing in the `USBDelay` function in USB software versions 1.0 and 1.0.1, it does works as described with USB version 1.1 and higher. For USB versions earlier that 1.1, the Notification Manager can be used to effect a transition to task level.

Polling Versus Asynchronous Completion (Important)

The Mac OS USB Service Library (USL) allows class drivers to poll for the completion of USL function calls by polling the `usbRefcon` field of the parameter block. In general, it is strongly advised to use asynchronous completion via the call back mechanism defined in "Asynchronous Call Support," instead of polling. Polling the `usbRefcon` field is discouraged and should only be implemented under exceptional circumstances.

The primary concern with polling lies with code that is designed to use the asynchronous USL call mechanism, and then enters a tight code loop where little happens except to check the `usbRefcon` field, and only exits the loop when the `usbRefcon` field changes. This form of polling robs time from the system, because nothing useful can happen while the code runs in the tight loop.

USB time scales are on the order of milliseconds. A tight polling loop represents and eternity of time wasted, when the system could be doing useful work. Some devices have very slow completion times. Completion times on the order of 100ms are not uncommon. If a driver polled for this length of time, the user would notice the pause, and system performance could suffer. In fact, there are circumstances in which polling the parameter block can cause the system to hang. These circumstances and some guidelines for avoiding them are further defined below:

Never poll from secondary interrupt time. Secondary interrupts are queued, and most I/O including the USL completes at secondary interrupt time. If you poll within secondary interrupt time, USL calls will never get a chance to complete, and the poll will never complete.

If you poll from task time the system may still hang. In order to guard against this you should either:

1). Give time back to the system by calling waitnext event.

2). Only poll for a limited time.

Option 1 is usually only practical from an application. Applications should not be making USL calls. Only class drivers should make USL calls. The use of USL calls by an application, is not supported.

Option 2 can be used by class drivers. The USB standard calls for a 5 second timeout on all transactions. However, this is not currently implemented by the USL. The polling software should make frequent calls to `USBGetFrameNumberImmediate` to determine the elapsed time. If the elapsed time becomes too great, the attempt should be abandoned with the `USBAbortPipeByReference` call.

In general, it is best to use asynchronous completion routines wherever possible.

USL Functions

This section describes the functions in the USB Services Library.

USB Configuration Functions

To make a connection to a USB device, the class driver must find an interface function that meets its requirements, and then configure the USB device for subsequent operations. The functions described in this section provide Mac OS USB configuration services.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

The first thing a class driver needs to do when configuring a device is find the *function* in a device configuration on which the driver is to operate and set the configuration. The *function* the driver is interested in is represented in the USB device hierarchy by an interface inside of a configuration. The programmatic view of the USB hierarchy is devices-> configurations-> interfaces-> endpoints.

USBFindNextInterface

The `USBFindNextInterface` function is used to find an interface and its parent configuration. The `USBFindNextInterface` function searches through all configurations for interfaces matching the USB class, subclass, and protocol input parameters and returns both the number of the configuration it found the interface in, and the number of the matching interface. The interface numbers are returned in the order in which they appear in the configuration descriptor. You can iterate through the list of both configurations and interfaces until you find the interface you are looking for. The returned configuration information is used in the `USBOpenDevice` function call to set the device configuration containing the interface.

```
OSStatus USBFindNextInterface(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBFindNextInterface` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Device reference
`-- usbBuffer`	Should be set to 0 (0 returned); reserved in this call
`-- usbActCount`	Should be set to 0 (0 returned); reserved in this call
`-- usbReqCount`	Should be set to 0 (0 returned); reserved in version 1.0 of the USBPB. Version 1.1 and later of the USB software supports passing in a maximum power requirement for the configuration, or 0 if not interested in the power

	requirement. If the configuration requires more power than specified, the `kUSBDevicePowerProblem` error is returned.
`--> usbFlags`	Should be set to 0 (0 returned); reserved in this call
`<--> usbClassType`	--> Class, 0 matches any class <-- Class value for interface found
`<--> usbSubclass`	--> Subclass, 0 matches any subclass <-- Subclass value for interface found
`<--> usbProtocol`	--> Protocol, 0 matches any protocol <-- Protocol value for interface found
`<--> usbWValue`	Configuration number; start with 0
`<--> usbWIndex`	--> Interface number; start with 0 <-- Interface number
`<--> usbOther`	Alternate interface, set to 0xff to find first alternate only

For alternate interface settings, the `usbOther` field provides a method for getting details about a specific alternate interface or all of the alternate interfaces as a set. For example, the compound class driver would find all the interfaces in a device and load drivers for those interfaces. It would, however, treat the set of alternates as one interface and load only one driver for the alternate. That alternate driver would then have to determine what alternate settings were appropriate and choose the appropriate driver for those settings.

To support finer granularity search criteria when looking for a specific interface in a device, and to avoid matching an interface that requires too much power, the `usbReqCount` field supports passing a value for the maximum power supported by the configuration. A driver can look for an interface with a specific class, subclass, and protocol in a configuration that supports less than a specified amount of power. It the appropriate amount of power is not available for the device, an error of `kUSBDevicePowerProblem` is retuned. The driver can choose to notify the user or continue looking for an interface that satisfies all the parameters.

If a driver chooses not to pass the power requirement in the `usbReqCount` field when looking for an interface, the USL matches interfaces with the other parameters even though they require more power than is available. When the configuration for a device that requires more power than is available is passed in with the `USBOpenDevice` function, the USL generates a power alert to the USB Manager.

The value of 0 for `usbClass`, `usbSubclass`, or `usbProtocol` is a wildcard value that indicates the caller is interested in whatever information can be found for those parameters in the search. The actual values for any interface found are returned in those fields. If a driver wants to make a subsequent call using wildcards for the class, subclass, and protocol values, a 0 value must be explicitly passed in for the `usbClass`, `usbSubclass`, and `usbProtocol` fields, since the actual values for the interface found during the last call are returned in those fields of the parameter block.

The `usbWValue` and `usbWIndex` fields should be set to 0 upon first entry of the `USBFindNextInterface` function to indicate the search should start at the beginning of the configuration descriptors. For subsequent calls to the `USBFindNextInterface` function, the values returned in the `usbWValue` and `usbWIndex` fields should be passed back in without modification. The next interface matching the specified values will be found.

Errors returned by the `USBFindNextInterface` function include

`paramErr`		`usbBuffer` pointer, `usbReqCount`, or `usbActCount` fields are not set to 0
`kUSBDevicePowerProblem`	-6976	an interface was found that matches the class, subclass, and protocol, but failed the power requirements.
`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBInternalErr`, `paramErr`	-6999	internal configuration descriptor cache corrupted
`kUSBNotFound`	-6987	interface or configuration specified is not in configuration descriptors

USBOpenDevice

Once a suitable interface is found, the device is opened with the configuration specified in the `USBOpenDevice` function.

```
OSStatus USBOpenDevice(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBOpenDevice` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine

`-->` `usbRefcon`	General-purpose value passed back to the completion routine
`-->` `usbReference`	Device reference
`--` `usbBuffer`	Should be set to 0 (0 returned); reserved in this call
`--` `usbActCount`	Should be set to 0 (0 returned); reserved in this call
`--` `usbReqCount`	Should be set to 0 (0 returned); reserved in this call
`-->` `usbWValue`	Configuration number
`-->` `usbFlags`	Should be set to 0
`<--` `usbOther`	Number of interfaces in configuration

The configuration number is an arbitrary number assigned by the device to label the configurations. The number is usually sequential 1,2,3 and so on, but not guaranteed to be so.

Errors returned by the `USBOpenDevice` function include

`paramErr`		`usbBuffer` pointer, `usbReqCount`, or `usbActCount` fields are not set to 0
`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBDevicePowerProblem`	-6976	the device requires more power than is currently available.
`kUSBDeviceBusy`	-6977	the device is already being configured
`kUSBInternalErr`, `paramErr`	-6999	internal configuration descriptor cache corrupted
`kUSBNotFound`	-6987	interface or configuration specified is not in configuration descriptors

Opening An Interface

The `USBNewInterfaceRef` function performs the first step in opening an interface. The `USBConfigureInterface` function (page 4-57) completes the process by setting up the interface for further communication.

USBNewInterfaceRef

The `USBNewInterfaceRef` function generates a new reference number that allows the interface in the specified device to be referred to. The interface reference can be used in most circumstances where a device reference can be used. An interface reference is required to be passed to the function `USBExpertInstallInterfaceDriver` (page 4-77).

```
OSStatus USBNewInterfaceRef(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBNewInterfaceRef` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbReference`	--> Device reference of device being configured <-- Interface reference returned
`-- usbBuffer`	Should be set to 0 (0 returned); reserved in this call
`-- usbActCount`	Should be set to 0 (0 returned); reserved in this call
`-- usbReqCount`	Should be set to 0 (0 returned); reserved in this call
`--> usbWIndex`	Interface number
`--> usbFlags`	Should be set to 0

If you create an interface reference, the interface reference must be disposed of in the driver finalize routine.

Errors returned by the `USBNewInterfaceRef` function include

`paramErr`		`usbBuffer` pointer, `usbReqCount`, or `usbActCount` fields are not set to 0
`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBDeviceBusy`	-6977	the device is already being configured
`kUSBNotFound`	-6987	interface or configuration specified is not in configuration descriptor
`kUSBOutOfMemoryErr`	-6988	ran our of internal structures

Configuring The Device Interface(s)

Configuring the interface or interfaces of a device is done with the `USBConfigureInterface` function.

USBConfigureInterface

The `USBConfigureInterface` function sets the interface on the device, and opens each pipe in the interface. The number of pipes opened is returned. It can also be used to set an alternate interface on the device.

This function does not currently operate as defined above. It does not set the device interface, it will in the future. At this time, the class driver must call the `USBDeviceRequest` function and make a `set_interface` device request to set the device interface. The driver can then call `USBConfigureInterface` to open the pipes in the interface and get the number of pipes. If required, an alternate interface can be specified upon entry in the `usbOther` field.

```
OSStatus USBConfigureInterface(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBConfigureInterface` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Interface reference obtained from `USBNewInterfaceRef`
`-- usbBuffer`	Should be set to 0 (0 returned); reserved in this call
`-- usbActCount`	Should be set to 0 (0 returned); reserved in this call
`-- usbReqCount`	Should be set to 0 (0 returned); reserved in this call
`--> usbFlags`	Should be set to 0
`<--> usbOther`	--> Alternate interface; <-- Number of pipes in interface

If information about an individual pipe or other element is needed, a device request has to be made.

Configuring an already opened interface is not an error. This sets the alternate and flags settings for the interface. It also invalidates any pipe references you are using.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Errors returned by the `USBConfigureInterface` function include

`kUSBUnknownInterfaceErr`	-6978	`usbReference` does not refer to a current interface
`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`paramErr`		`usbBuffer` pointer, `usbReqCount`, or `usbActCount` fields are not set to 0
`kUSBInternalErr, paramErr`	-6999	internal configuration descriptor cache corrupted
`kUSBNotFound`	-6987	interface or configuration specified is not in configuration descriptor
`kUSBIncorrectTypeErr`	-6995	interface has control endpoints
`kUSBTooManyPipesErr`	-6996	ran out of internal structures

Finding A Pipe

After the functions used to open the interface have completed, you need to work out which already open pipe in the interface is the one you want to communicate through.

USBFindNextPipe

The `USBFindNextPipe` function can be used to either find a specific pipe, as specified by the direction in the `usbFlags` field and type in the `usbClassType` field, or to search through the available pipes.

```
OSStatus USBFindNextPipe(USBPB *pb);
```

Required fields required the `USBPB` parameter block for the `USBFindNextPipe` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbReference`	--> Interface or pipe reference <-- Pipe reference

-- `usbBuffer`	Should be set to 0 (0 returned); reserved in this call
-- `usbActCount`	Should be set to 0 (0 returned); reserved in this call
-- `usbReqCount`	Should be set to 0 (0 returned); reserved in this call
<--> `usbFlags`	--> Specific direction of pipe (`kUSBIn` or `kUSBOut`) or `kUSBAnyDirn` as a wildcard <-- Direction of input or output pipe
<--> `usbClassType`	--> Specific endpoint type (`kUSBControl`, `kUSBInterrupt`, or `kUSBBulk`) or `kUSBAnyType` as a wildcard <-- Endpoint type
<-- `usbWValue`	Maximum packet size of endpoint

This function takes either an interface or pipe reference in the `usbReference` field. To find the first pipe, make a call to the function with an interface reference. To find the next pipe, enter the pipe reference returned by the previous call.

The `usbFlags` field takes either a specified endpoint direction or a wildcard of `kUSBAnyDirn`. The `usbClassType` field takes either a specified endpoint type or a wildcard of `kUSBAnyType`. For example, if you specify values for an input interrupt pipe, the function returns only the input interrupt pipes found. If a wildcard is used, all pipes of any type and direction found are returned.

Errors returned by the `USBFindNextPipe` function include

`paramErr`		`usbBuffer` pointer, `usbReqCount`, or `usbActCount` fields are not set to 0
`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown
`kUSBNotFound`	-6987	interface or configuration specified is not in configuration descriptor

Getting Information About an Open Interface or Pipe

Information about an opened interface or pipe is contained within the interface and pipe descriptors, and other descriptors associated with them.

USBFindNextAssociatedDescriptor

You use the `USBFindNextAssociatedDescriptor` function to find a specific interface or pipe descriptor, or any descriptor associated with the interface or endpoint. For example, a HID interface driver could use this function to find HID descriptors.

```
OSStatus USBFindNextAssociatedDescriptor(USBPB *pb);
```

Required fields required the `USBPB` parameter block for the `USBFindNextAssociatedDescriptor` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Interface or pipe reference
`<--> usbWIndex`	Descriptor index start at zero
`--> usbBuffer`	Descriptor buffer
`--> usbReqCount`	Size of buffer
`<-- usbActCount`	Size of the descriptor returned
`<--> usbOther`	Descriptor type (a value of 0 matches any)

The `USBFindNextAssociatedDescriptor` function steps through the descriptors following the relevant interface or endpoint descriptor, and returns the descriptors matching the given parameters. If `usbReference` is an interface reference, all the descriptors are returned until the next interface descriptor is found, or until the end of the configuration descriptor is reached. If `usbReference` is a pipe reference, all of the descriptors are returned until the next endpoint or interface descriptor is found, or until the end of the configuration descriptor is reached.

The `usbWIndex` field provides an index into all the available descriptors. A value of 1 describes the interface or endpoint descriptor itself, so passing 0 allows the interface or endpoint descriptor to be returned. If `usbWIndex` is passed back in the next call untouched, the function returns the next available matching descriptor.

The `usbOther` field contains a descriptor type to match. If searching for any type (`usbOther` set to 0) all descriptors are matched. To use this method of search again for all descriptors, the usbOther field has to be set to 0 each time the function is called.

The errors returned by the USBFindNextAssociatedDescriptor function include:

kUSBUnknownInterfaceErr	-6978	usbReference does not refer to a current interface or pipe
kUSBInternalErr, paramErr	-6999	internal configuration descriptor cache corrupted
kUSBNotFound	-6987	interface or configuration specified is not in configuration descriptor

USBDisposeInterfaceRef

The USBDisposeInterfaceRef function closes the specified interface currently opened. The interface reference obtained with the USBNewInterfaceRef function for this interface is no longer valid after the call USBDisposeInterfaceRef call completes.

```
OSStatus USBDisposeInterfaceRef(USBPS *pb);
```

Required fields in the USBPB parameter block for the USBDisposeInterfaceRef function are

--> pbLength	Length of parameter block
--> pbVersion	Parameter block version number
--> usbCompletion	The completion routine
--> usbRefcon	General-purpose value passed back to the completion routine
--> usbReference	Interface reference for the interface to close.
--> usbFlags	Should be set to 0

If the usbCompletion field is set to kUSBNoCallBack, the call back mechanism is not invoked. This is useful for finalization routines which need to clean up immediately and can't wait for a callback routine to complete.

If the no call back option (kUSBNoCallBack) is used, the parameter block is free as soon as the USBDisposeInterfaceRef call returns.

Errors returned by the USBDisposeInterfaceRef function include

`kUSBUnknownInterfaceErr`	-6978	`usbReference` does not refer to a current interface

Generalized USB Device Request Function

The USB standard specifies one of the fields of a control request as a `BMRequestType`. This field is a bit-mapped byte that tells the USB function about the request. Information about the request includes direction of data flow, how the function is defined (standard, class, or vendor specific) and what logically is the recipient of the request.

USBMakeBMRequestType

The `USBMakeBMRequestType` function formats device and control request type parameters into the `bmRequestType` format, which are passed to the USL in the `usbBMRequestType` field of the `USBDeviceRequest` function.

The `USBMakeBMRequestType` function returns a UInt8 or 0xff if one or more of the parameters is incorrect. A value of 0xff is not a legal value and is not accepted by the subsequent control call.

```
OSStatus USBMakeBMRequestType(UInt8 direction, UInt8 type,
                              UInt8 recipient);
```

`direction`	Direction of data flow, kUSBOut, kUSBIn, or kUSBNone
`type`	Definition of the request, kUSBStandard, kUSBClass, or kUSBVendor
`recipient`	Part of the device receiving the request, kUSBDevice, kUSBInterface, kUSBEndpoint, or kUSBOther

All USB devices respond to requests from the host on the device's default pipe. These requests are made using control transfers. The request and the request's parameters are sent to the device in the setup packet.

USBDeviceRequest

The `USBDeviceRequest` function performs control transactions to default pipe 0 (zero) of a device.

```
OSStatus USBDeviceRequest(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBDeviceRequest` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	The device reference passed to the driver when it is loaded
`--> usbBMRequestType`	The `usbBMRequestType` field is made up of the `direction`, `type`, and `recipient` values

`direction` One of the following:

`kUSBIn`	Data will be transferred to the host.
`kUSBOut`	Data will be transferred to the device.
`kUSBNone`	No data will be transferred. The length and buffer parameters are ignored.

`type` One of the following:

`kUSBStandard`	A request defined in the USB standard.
`kUSBClass`	A request defined in a class standard.
`kUSBVendor`	A vendor unique request type.

`recipient` One of the following:

`kUSBDevice`	The request is to the whole device.
`kUSBInterface`	The request is to a specific interface in the device.
`kUSBEndpoint`	The request is to a specific pipe endpoint in a device.
`kUSBOther`	The request is going somewhere else.

`--> usbBRequest`	Defined by the USB standard, defined by a class driver standard, or vender unique. See "`usbBRequest Constants" (page 4-98)`
`--> usbWValue`	General parameter unique to the transaction request. This value is in host endian format, and will be swapped if necessary when it is sent to the device.
`--> usbWIndex`	General parameter unique to the transaction request. This value is in host endian format, and will be swapped if necessary when it is sent to the device.
`--> usbReqCount`	Specifies the size of the data to transfer. If this is set to 0, no transfer occurs
`--> usbBuffer`	Points to the data to be transferred (`kUSBOut` request) or where data will end up (`kUSBIn` request). The buffer should be at least as big as the size specified in the `usbReqCount` field. If `usbBuffer` is set to `nil`, no data is transferred regardless of the value of length
`<-- usbActCount`	Specifies the actual amount of data transferred on completion
`--> usbFlags`	Should be set to 0, or `kUSBAddressRequest` for control transactions addressed to an interface or endpoint

The request is sent to the default pipe 0 and the relevant data is transferred.

The flag `kUSBAddressRequest` supports USB control transactions addressed to an interface or endpoint of a device. The `kUSBAddressRequest` flag allows the control call to be made without the driver explicitly knowing the number of the endpoint or interface before the call is made. The USL fills in the interface or endpoint number in the setup packet based on the pipe or interface reference that is passed in with the call.

To use the addressed device request feature, specifiy the `kUSBAddessRequest` flag in the `usbFlags` field. If the `recipient` field of the `BMRequestType` is an endpoint or interface, the relevant endpoint or interface number is derived from the pipe or interface reference that is passed in the `usbReference` field. The interface or endpoint number is put into the `WIndex` field of the setup packet before the control transaction call takes place.

For additional information about the parameters specified for control transactions, see section 9 of the USB Specification.

If the request is a `set_config` request, the `USBDeviceRequest` function returns the same errors as those for the `USBOpenDevice` function. Other errors returned by the `USBDeviceRequest` function include

`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBRqErr`	-6994	the value in the `bmRequestType` field is not valid
`kUSBUnknownRequestErr`	-6993	request code for a standard USB call is not recognized

USB Transaction Functions

There are four transaction types supported by the USL, control, interrupt, bulk, or isochronous. When making isochronous calls, you set the `pbVersion` field in the `USBPB` parameter block to `kUSBIsocPBVersion`, and use the isochronous variant of the `USBPB` parameter block, described in "The Isochronous Version Of The USBPB" (page A-119).

When making function calls that are not a direct result of a call to the Device Manager, such as the USB transaction functions, class drivers should arrange to hold the memory for the data buffer used in the `usbBuffer` field. The driver can do this with either the `PrepareMemoryForIO` function described in *"Designing PCI Cards & Drivers for Power Macintosh Computers,"* or the older `HoldMemory` function described in the Virtual Memory Manager section of *"Inside Macintosh:Memory."*

The `PrepareMemoryForIO` and `HoldMemory` functions are only safe to call at task level (any non-secondary interrupt time). One way to safely prepare the memory is to create and hold a single buffer at driver initialization time, which occurs at task level. You then use this buffer to make all transaction requests. The USB Manager holds the driver code and globals automatically when the driver is loaded. Therefore, declaring a buffer in the global space guarantees the memory will be held before requesting a transaction.

USBIntRead

The `USBIntRead` function queues an interrupt transaction on the specified pipe. The device is periodically polled and the transaction completes when the device

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

returns some data. This is a read operation only. Version 1.0 of the USB specification does not define an interrupt write function.

```
OSStatus USBIntRead(USBPB *pb);
```

Required fields in the USBPB parameter block for the USBIntRead function are

-->	pbLength	Length of parameter block
-->	pbVersion	Parameter block version number
-->	usbCompletion	The completion routine
-->	usbRefcon	General-purpose value passed back to the completion routine
-->	usbReference	The pipe reference returned by the USBFindNextPipe function
-->	usbReqCount	Specifies the size of the data to transfer. If this is set to 0, anything but a zero length packet causes an error
-->	usbBuffer	Points to a buffer to which the incoming data is transferred
<--	usbActCount	Specifies the actual amount of data transferred on completion
-->	usbFlags	Should be set to 0

In order to avoid the loss of data when transferring data from a device, usbBuffer and usbReqCount should be a multiple of the value of the MaxPacketSize field, in the device's endpoint descriptor.

Errors returned by the USBIntRead function include

kUSBUnknownPipeErr	-6997	pipe reference specified is unknown
kUSBIncorrectTypeErr	-6995	pipe is not an interrupt pipes
kUSBPipeIdleErr	-6980	specified pipe is in the idle state
kUSBPipeStalledErr	-6979	specified pipe is stalled

USBBulkRead

The USBBulkRead function can be used to request multiple bulk transactions on an inbound bulk pipe to fulfill the size of request specified, or for the entire transfer.

```
OSStatus USBBulkRead(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBBulkRead` function are

`-->` `pbLength`	Length of parameter block
`-->` `pbVersion`	Parameter block version number
`-->` `usbCompletion`	The completion routine
`-->` `usbRefcon`	General-purpose value passed back to the completion routine
`-->` `usbReference`	The pipe reference returned by the USBFindNextPipe function
`-->` `usbReqCount`	Specifies the size of the data to transfer. Must be a multiple of the packet size. If it is not a multiple of the packet size, the last packet my overrun. If set to 0, any non-zero size transfer causes an error
`-->` `usbBuffer`	Points to a buffer to which the incoming data is transferred
`<--` `usbActCount`	Specifies the actual amount of data transferred on completion
`-->` `usbFlags`	Should be set to 0

In order to avoid the loss of data when transferring data from a device, `usbBuffer` and `usbReqCount` should be a multiple of the endpoint `MaxPacketSize` in the device's endpoint descriptor.

Errors returned by the `USBBulkRead` function include

`kUSBUnknownPipeErr`	`-6997`	pipe reference specified is unknown
`kUSBIncorrectTypeErr`	`-6995`	pipe reference is not a bulk-in pipe
`kUSBPipeIdleErr`	`-6980`	specified pipe is in the idle state
`kUSBPipeStalledErr`	`-6979`	specified pipe is stalled

USBBulkWrite

The `USBBulkWrite` function requests multiple bulk out transactions on an outbound bulk pipe to fulfill the size of request specified.

```
OSStatus USBBulkWrite(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBBulkWrite` function are

`-->` `pbLength`	Length of parameter block
`-->` `pbVersion`	Parameter block version number

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

`-->` `usbCompletion`	The completion routine
`-->` `usbRefcon`	General-purpose value passed back to the completion routine
`-->` `usbReference`	The pipe reference returned by the `USBFindNextPipe` or `USBOpenPipe` functions
`-->` `usbReqCount`	Specifies the size of the data to transfer. If this is set to 0, no data transfer occurs, but the device senses a 0 length bulk transaction
`-->` `usbBuffer`	Points to the data to be transferred. The buffer should be at least as big as the size specified in the `usbReqCount` field. If the buffer is set to `nil`, no data is transferred regardless of the value of `usbReqCount`
`<--` `usbActCount`	Specifies the actual amount of data transferred on completion
`-->` `usbFlags`	Should be set to 0

Errors returned by the `USBBulkWrite` function include

`kUSBUnknownPipeErr`	`-6997`	pipe reference specified is unknown
`kUSBIncorrectTypeErr`	`-6995`	pipe reference is not a bulk-out pipe
`kUSBPipeIdleErr`	`-6980`	specified pipe is in the idle state
`kUSBPipeStalledErr`	`-6979`	specified pipe is stalled

USBIsocRead

The `USBIsocRead` function is defined as follows:

```
OSStatus USBIsocRead(USBIsocPB *pb);
```

Required fields in the isochronous version of the `USBPB` parameter block for the `USBIsocRead` function are

`-->` `pbLength`	Length of parameter block
`-->` `pbVersion`	Parameter block version, must be `kUSBIsocPBVersion` for isochronous function calls
`<--usbStatus`	Aggregate status, see discussion in “The Isochronous Version Of The USBPB” (page A-119)
`-->` `usbCompletion`	The completion routine

`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	The isochronous pipe reference returned by the `USBFindNextPipe` function
`--> usbReqCount`	Specifies the size of the data to transfer. May be any size, but no less than the sum of the individual packets sizes. If set to 0, any non-zero size transfer causes an error.
`--> usbBuffer`	Points to a buffer to which the incoming data is transferred
`<-- usbActCount`	Specifies the actual amount of data transferred on completion
`--> usbFlags`	Should be set to 0
`--> usb.isoc.FrameList`	Pointer to the list of frame structures
	`--> frReqCount` Number of bytes requested for each packet
	`<-- frStatus` Status returned by packet
	`<-- frActCount` Actual bytes transferred by packet
`--> usb.isoc.NumFrames`	Number of frames to attempt transfers in
`--> usbFrame`	Frame number of the first frame to transfer data

USBIsocWrite

The `USBIsocWrite` function is defined asfollows:

```
USBIsocWrite(USBIsocPB *pb);
```

Required fields in the isochronous version of the `USBPB` parameter block for the `USBIsocWrite` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version, must be `kUSBIsocPBVersion` for isochronous function calls
`<--usbStatus`	Aggregate status, see discussion in “The Isochronous Version Of The USBPB” (page A-119)
`--> usbCompletion`	The completion routine

`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	The isochronous pipe reference returned by the `USBFindNextPipe` or `USBOpenPipe` functions
`--> usbReqCount`	Specifies the size of the data to transfer. If this is set to 0 in the frReqCount field in the frame structure, the call sends a packet size of 0.
`--> usbBuffer`	Points to the data to be transferred. The buffer should be at least as big as the size specified in the `usbReqCount` field. If the buffer is set to `nil`, no data is transferred regardless of the value of `usbReqCount`
`<-- usbActCount`	Specifies the actual amount of data transferred on completion
`--> usbFlags`	Should be set to 0
`--> usb.isoc.FrameList`	Pointer to the list of frame structures

`--> frReqCount`	Number of bytes requested for each packet
`<-- frStatus`	Status returned by packet
`<-- frActCount`	Actual bytes transferred by packet

`--> usb.isoc.NumFrames`	Number of frames to attempt transfers in
`--> usbFrame`	Frame number of the first frame to transfer data

Pipe State Control Functions

A pipe's state is governed by two factors:

- the state of the device's endpoint
- the USL's state

The USL state can be one of the following:

- Active: The pipe is open and can transmit data.
- Stalled: An error occurred on the pipe, no new transactions are accepted until the stall is cleared.
- Idle: The pipe will not accept any transactions.

A transaction error (errors -6915 to -6901) causes the pipe to enter the stalled state. The class driver can change the state of the pipe using the functions in this section.

Note that the pipe and interface control functions differ from most other USL calls in these two ways:

- They do not take a parameter block as a parameter.
- They complete synchronously. There is no facility for asynchronous completion.

Also note that pipe 0 to a device cannot be stalled. If a communication error happens on pipe 0, the stall is automatically cleared before the call completes. Thus some of these functions can affect a device's default pipe 0 and some can't. Those functions that operate on both the default pipe 0 and pipes other than pipe 0, take a device reference for the default pipe or a pipe reference for a specific pipe. Those functions that can't affect the default pipe, take only a pipe reference.

These calls can be used on a device's default pipe 0:

```
USBGetPipeStatusByReference
USBAbortPipeByReference
USBResetPipeByReference
```

These calls cannot be used on a device's default pipe 0:

```
USBClearPipeStallByReference
USBSetPipeIdleByReference
```

Except for entering the stalled state on an error, the USL does not keep track of the state of the device's endpoint. The class driver must keep track of the state of the endpoint.

Data Toggle Synchronization

When a pipe is reset, aborted, or had a stall cleared, the expected data toggle on that pipe's endpoint is reset to data0. This means that the next packet read on that pipe may be discarded unless the device is told to synchronize its endpoint data toggle.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

The method of synchronizing the endpoint for the device is device specific. In general, it should be possible to perform endpoint data toggle synchronization with a call to the `USBDeviceRequest` function addressed to the endpoint in question. A USB device request command of `CLEAR_FEATURE` and a feature selector of `ENDPOINT_STALL` should complete the required data toggle synchronization.

USBGetPipeStatusByReference

The `USBGetPipeStatusByReference` function returns status on a specified pipe or the device's default pipe 0.

```
OSStatus USBGetPipeStatusByReference(USBReference ref,
                          USBPipeState *state);
```

`ref`	Pipe reference.
`state`	Returns the pipe state, it can be one of these constants:

`kUSBActive`	Pipe can accept new transactions
`kUSBIdle`	Pipe cannot accept new transactions
`kUSBStalled`	An error occurred on the pipe

Errors returned by the `USBGetPipeStatusByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown

In version 1.0 of the USB Services software the `USBGetPipeStatusByReference` function does not operate as defined above. If the pipe is not active, it returns an error and the state is not set. The `USBGetPipeStatusByReference` function does work as defined in version 1.0.1 and later of the USB Services software. If `noErr` is returned, the state is returned correctly. If this call returns an error, the error should be examined to see what state the pipe is in.

Errors returned by the `USBGetPipeStateByReference` function include:

`noErr`	0	specified pipe is active
`kUSBPipeIdleErr`	-6980	specified pipe is in the idle state
`kUSBPipeStalledErr`	-6979	specified pipe is stalled

USBAbortPipeByReference

The `USBAbortPipeByReference` function aborts operations on a specified pipe or the device's default pipe 0.

```
OSStatus USBAbortPipeByReference(USBReference ref);
```

`ref`	Pipe reference, or device reference for implicit default pipe 0.

All outstanding transactions on the pipe are returned with a `kUSBAborted` status. The state of the pipe is not affected.

After this function is called, the device's endpoint needs to be synchronized with the host's endpoint. See "Data Toggle Synchronization" (page 4-71) for information about how to accomplish endpoint data toggle synchronization.

Errors returned by the `USBAbortPipeByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown

USBResetPipeByReference

The `USBResetPipeByReference` function resets the specified pipe or the device's default pipe 0.

```
OSStatus USBResetPipeByReference(USBReference ref);
```

`ref`	Pipe reference, or device reference for implicit default pipe 0.

All outstanding transactions on the pipe are returned with a `kUSBAborted` status. The pipe status is set to active. The stalled and idle state are cleared.

After this function is called, the device's endpoint needs to be synchronized with the host's endpoint. See "Data Toggle Synchronization" (page 4-71) for information about how to accomplish endpoint data toggle synchronization.

IMPORTANT

For USB parameter block version 1.0, the implementation of `USBResetPipeByReference` does nothing if passed a real pipe reference. However, if the function is passed a non-existant pipe reference, it will corrupt low memory. Version 1.0.1 and later of the USB Services software corrects this problem.

Errors returned by the `USBRestPipeByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown

In version 1.0 of the USB Services software, the pipe may or may not have been made active, depending on whether the pipe was previously stalled or not, and the `kUSBPipeIdleErr` is returned. If an idle pipe was not stalled, it is not affected. If an idle pipe was stalled, it is made active. In version 1.0.1 and later of the USB Services software this behavior is corrected.

In version 1.0 of the USB Services software, the `kUSBPipeStalledErr` is returned if the pipe was previously idle and the call succeeded despite the error. This behavior is not an error and noErr is returned in verions 1.0.1 and later of the USB Services software.

`kUSBPipeStalledErr`	-6979	pipe stalled, pipe is reset despite the error

USBClearPipeStallByReference

The `USBClearPipeStallByReference` function clears a stall on the specified pipe. This call can only be used on a pipe, not on a device's default pipe 0.

```
OSStatus USBClearPipeStallByReference(USBPipeRef ref);
```

`ref`	Pipe reference.

All outstanding transactions on the pipe are returned with a `kUSBAborted` status. The pipe status is set to active; if the pipe was previously idle it is set back to idle. The stalled state is cleared, idle is not.

A call to this function does not clear a device's endpoint stall. The class driver has to take care of that by using USB standard device commands, such as `CLEAR_ENDPOINT_STALL`. The class driver may need to take other remedial actions.

After this function is called, the device's endpoint needs to be synchronized with the host's endpoint. See "Data Toggle Synchronization" for information about how to accomplish endpoint data toggle synchronization.

Errors returned by the `USBClearPipeStallByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown
`kUSBPipeIdleErr`	-6980	specified pipe is in the idle state

USBSetPipeIdleByReference

The `USBSetPipeIdleByReference` function sets a specified pipe to the idle state. This call can be used only on a specified pipe, not on a device's default pipe 0.

```
OSStatus USBSetPipeIdleByReference(USBPipeRef ref);
```

`ref`	Pipe reference.

The state of the pipe is set to idle. No outstanding transactions are affected.

Errors returned by the `USBSetPipeIdleByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown

In version 1.0 of the USB Services software, the following errors are returned if the pipe is not currently active. In these instances, the call has succeeded despite the returned error. This behavior is not an error and noErr is returned in verions 1.0.1 and later of the USB Services software.

`kUSBPipeIdleStalled`	-6979	pipe was stalled, pipe is still active despite error
`kUSBPipeIdleErr`	-6980	specified pipe is in the idle state

USBSetPipeActiveByReference

The `USBSetPipeActiveByReference` function sets the state of a specified pipe to active.

```
OSStatus USBSetPipeActiveByReference(USBPipeRef ref);
```

`ref`	Pipe reference.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

The pipe status is set to active if the pipe is not stalled. The idle state is cleared, stalled is not.

Errors returned by the `USBSetPipeActiveByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference specified is unknown
`kUSBPipeIdleStalled`	-6979	pipe was stalled, pipe is set idle

In version 1.0 of the USB Services software, the following error is returned if the pipe was previously idle. In this instance the call has succeeded despite the returned error. This behavior is not an error and `noErr` is returned in verions 1.0.1 and later of the USB Services software.

`kUSBPipeIdleErr`	-6980	pipe was previously idle, pipe is still made active

USB Management Services Functions

The USL provides an interface to services provided by the USB Manager. These services make it so class drivers need only link against the USB Services library or driver services library.

The errors returned by the USB Management functions include:

`kUSBBadDispatchTable`	-6950	improper driver dispatch table
`kUSBUnknownNotification`	-6949	notification type not defined
`kUSBQueueFull`	-6948	internal queue full

USBExpertInstallDeviceDriver

The `USBExpertInstallDeviceDriver` function notifies the USB Manager that there is a device that needs a driver matched and loaded. Typically only hub drivers need the service provided by this function.

```
OSStatus USBExpertInstallDeviceDriver(USBDeviceRef ref,
                        USBDeviceDescriptorPtr *desc
                        USBReference hubRef,
                        UInt32 port,
                        UInt32 busPowerAvailable);
```

The `ref` parameter can be a device reference or an interface reference. Similarly the `desc` parameter can be a device or interface descriptor.

`ref`	Device reference of the new device.
`desc`	Device descriptor of the device to find a driver for.
`hubRef`	The device reference of the parent hub of this device.
`port`	The parent port of this device.
`busPowerAvailable`	How much current is available from the bus for the device, in 2 milliamperes (mA) units. This should have one of two values, 100mA for a bus-powered hub parent and 500mA for a self-powered parent.

USBExpertRemoveDeviceDriver

The `USBExpertRemoveDeviceDriver` function notifies the USB Manager that a device has been removed from the bus and that the class driver for that device needs to be terminated. Typically only hub drivers need the service provided by this function.

```
OSStatus USBExpertRemoveDeviceDriver(USBDeviceRef ref);
```

The `ref` parameter can be a device reference or an interface reference.

`ref`	Device reference of the device removed from the bus.

USBExpertInstallInterfaceDriver

The `USBExpertInstallInterfaceDriver` function notifies the USB Manager that a class driver needs to be loaded for the given interface of the given device. This function is used by class drivers that select configurations and interfaces. The drivers that use this functionality are typically composite class drivers.

```
OSStatus USBExpertInstallInterfaceDriver(USBDeviceRef ref,
                    USBDeviceDescriptor *desc,
                    USBInterfaceDescriptor *interface,
                    USBReference hubRef,
                    UInt32 busPowerAvailable);
```

`ref`	Device reference of device containing the interface. (Note that this will eventually become an interface reference.)
`desc`	Device descriptor of the interface to find a driver for.
`interface`	Interface descriptor of interface to find a driver for.
`hubRef`	The device reference for the device containing this interface. Usually a device reference of a hub.
`busPowerAvailable`	How much current is available from the bus for the device, in 2 milliamperes (mA) units. This should have one of two values, 100mA for a bus-powered hub parent and 500mA for a self-powered parent.

USBExpertRemoveInterfaceDriver

The `USBExpertRemoveInterfaceDriver` Function notifies the USB Manager that a device has been removed from the bus and that the class driver needs to be disposed.

```
OSStatus USBExpertRemoveInterfaceDriver(USBInterfaceRef ref);
```

`ref`	Interface reference from the removed device

USB Time Utility Functions

This section describes the functions for managing time within the context of USB frames. A USB frame is approximately a 1 ms unit of time. Approximately, because it may vary a few bit times.

USBDelay

The `USBDelay` function calls back through the normal completion mechanism when the specified number of frames have passed. There is up to an extra one frame delay to accommodate synchronizing with USB frames. For example, 0 frames delay means after the current frame, which could be up to 1 ms plus any other system delays.

```
OSStatus USBDelay(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBDelay` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	A device, interface, or pipe reference which associates the call with a device
`--> usbReqCount`	Number of frames to delay
`<-- usbActCount`	Frame number at completion of delay
`--> usbFlags`	Callback at task time (`kUSBTaskTimeFlag`)

The `usbFlags` parameter can be used to request a call back at task time. A requested delay of `kUSBNoDelay` causes the call back to occur as soon as possible during system task time (as opposed to secondary interrupt time). Thus effecting a transition to task time.

There must be a valid USBReference passed in the `usbReference` field of the parameter block. If a `nil` value or a reference that does not match an existing device interface or pipe is passed in, the call returns immediately with an unknown device error.

If the device associated with this call is unplugged and its driver removed while this function call is pending, the function will not complete.

The `USBDelay` function returns the following error:

`kUSBUnknownDeviceErr`	`-6998`	`usbReference` does not refer to a current device

USBGetFrameNumberImmediate

The `USBGetFrameNumberImmediate` function returns the current frame number for the specified device. The function completes synchronously and is the recommended function to use for making time calculations for a class driver. It can be called at any execution level. This function also supports multiple USB bus implementations.

```
OSStatus USBGetFrameNumberImmediate(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBGetFrameNumberImmediate` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Device, interface, or endpoint reference
`--> usbReqCount`	Size of buffer (0 or size of UInt64)
`--> usbBuffer`	`Nil` or pointer to a UInt64 structure for full 64 bits of frame data.
`<-- usbActCount`	Size of data returned
`<-- usbFrame`	Low 32 bits of the current frame number

In multiple USB bus configurations, each bus has an independent frame count. The `USBGetFrameNumberImmediate` function takes any device, interface, or endpoint reference as input and returns the current frame number for the bus on which that device, interface, or endpoint is connected.

The frame count for each bus is maintained internally by the USB software as a 64 bit value. The `USBGetNextFrameNumberImmediate` function allows a driver to get either the low 32 bits of this value in the parameter block, or the full 64 bit value in a UInt64 structure. To get the low 32 bits, specify a value of nil in `usbBuffer` and a value of 0 in `usbReqCount`. To get the full 64 bits, specify the size of the UInt64 structure in the `usbReqCount` field and pointer to an address of the structure in `usbBuffer`.

This function does not call the completion routine. However, a value is required in the `usbCompletion` field. `kUSBNoCallBack` can be specified as the completion routine.

The `USBGetNextFrameNumberImmediate` function returns the following error:

`kUSBUnknownDeviceErr`	`-6998`	`usbReference` does not refer to a current device

USB Memory Functions

The memory functions allow USB class drivers to allocate and deallocate memory. Since memory allocation must typically occur at task time, the memory functions will queue the request until task time is available, then allocate the memory and return asynchronously. These functions are the preferred way of specifying memory requirements, because they relieve the class driver from monitoring execution levels when performing memory management functions.

USBAllocMem

The `USBAllocMem` function allocates a specified amount of memory.

```
OSStatus USBAllocMem(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBAllocMem` function are

`-->`	`pbLength`	Length of parameter block
`-->`	`pbVersion`	Parameter block version number
`-->`	`usbCompletion`	The completion routine
`-->`	`usbRefcon`	General-purpose value passed back to the completion routine
`-->`	`usbReference`	A device, interface, or pipe reference which associates the call with a device
`-->`	`usbReqCount`	Amount of memory required to be allocated
`<--`	`usbActCount`	Amount of memory actually allocated
`<--`	`usbBuffer`	Memory allocated
`-->`	`usbFlags`	Should be set to 0

There must be a valid `USBReference` passed in the `usbReference` field of the parameter block. If a `nil` value or a reference that does not match an existing device, interface, or pipe is passed in, the call returns immediately with an unknown device error.

If the device associated with this call is unplugged and its driver removed while this function call is pending, the function will not complete.

The `USBAllocMem` function returns the following error:

`kUSBUnknownDeviceErr`	`-6998`	`usbReference` does not refer to a current device

USBDeallocMem

The `USBDeallocMem` function deallocates the memory allocated with the `USBAllocMem` function.

```
OSStatus USBDeallocMem(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBDeallocMem` function are:

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	A device, interface, or pipe reference which associates the call with a device
`<--> usbBuffer`	--> previously allocated memory to be deallocated <-- pointer set to nil
`--> usbFlags`	Should be set to 0

You can pass `kUSBNoCallBack` as the `usbCompletion` field parameter to notify the USL that you want the operation to complete immediately if at task time. It is an error to specify no call back, if the current execution level is not task time.

If the `usbCompletion` field is set to `kUSBNoCallBack`, the call back mechanism is not invoked. This is useful for finalization routines which need to clean up immediately and can't wait for a callback routine to complete.

There must be a valid `USBReference` passed in the `usbReference` field of the parameter block. If a nil value or a reference that does not match an existing device, interface, or pipe is passed in, the call returns immediately with an unknown device error.

If the device associated with this call is unplugged and its driver removed while this function call is pending, the function will not complete.

The `USBDeAllocMem` function returns the following error:

`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBCompletionError`	-6984	`kUSBNoCallBack` was specified and current execution level is not task time

Byte Ordering (Endianism) Functions

There are two functions to deal with the differences in byte ordering between the Intel platform and Mac OS platform. The USB uses Intel byte ordering (called little endian) on all multibyte fields, which is reversed from the Mac OS byte ordering (called big endian, because the most significant byte appears at the lowest memory address). These functions are of endian neutral form. Using these functions correctly allows the code to be recompiled on an Intel endian platform and still work as expected.

All parameters and parameter block elements are automatically swapped by the USB Services Library. These functions need be used only for data that the USL has no knowledge of. This includes all descriptors returned from the descriptor functions.

If you need to embed a 16-bit USB constant in your code, you can use this macro:

```
USB_CONSTANT16(x)
```

`x`	The USB constant

This macro is only useful for the C or C++ programming languages.

HostToUSBWord

The `HostToUSBWord` function changes the byte order of a value from big endian to little endian.

```
UInt16 HostToUSBWord(UInt16 value)
```

USBTOHostWord

The `USBToHostWord` function changes the byte order of a value from little endian to big endian.

```
UInt16 USBToHostWord(UInt16 value)
```

If you need to embed a 16-bit USB constant in your code, you can use this macro:

```
USB_CONSTANT16(x)
```

`x`	The USB constant

This macro is only useful for the C or C++ programming languages.

USL Logging Services Functions

The USB Manager provides services to log status messages from drivers to aid in debugging and software development. The USL provides an interface to this service. When one of these messages is sent, it currently ends up in a buffer that the USB Prober utility knows how to read. Choose the USB Expert Log menu item in the USB Prober Window menu to look at the message.

USBExpertStatus

The `USBExpertStatus` function sends a general message out to the user. No weight is attached to this message by the operating system.

```
OSStatus USBExpertStatus(USBDeviceRef ref, void *pointer, UInt32 value);
```

`ref`	Device reference for the device driver giving notification.
`pointer`	A pointer to a string to display.
`value`	An arbitrary number to display.

USBExpertFatalError

The `USBExpertFatalError` function is intended to inform the system of nonrecoverable errors in a class driver. Currently no action is taken when this message is received. In the future it may cause a driver to be unloaded.

```
OSStatus USBExpertFatalError(USBDeviceRef ref, OSStatus status, void
                      *pointer, UInt32 value);
```

`ref`	Device reference for the device driver giving notification.
`status`	The error status that explains the failure.
`pointer`	A pointer to a error status string to display.
`value`	An arbitrary number to display.

USB Descriptor Functions

All of the USB configuration services were not fully implemented in earlier versions of the USL. USB configuration had to be performed manually by the class driver. To make this process less cumbersome, configuration descriptor parsing functions were provided. These functions are still available, and some sample drivers may use them, but it is recommended that you use the configuration services described in "USL Functions" (page 4-51).

The immediate functions (those that end with `Immediate` in the function name) may be used repeatedly with the same parameter block to search for interface and endpoint descriptors.

USBGetFullConfigurationDescriptor

The `USBGetFullConfigurationDescriptor` function returns the entire block of configuration data from the specified device and any associated descriptors, which includes interface and endpoint descriptors, and all of the information that pertains to them. The configuration data returned by the `USBGetFullConfigurationDescriptor` function is suitable for use with the

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

`USBFindNextInterfaceImmediate` and the `USBFindNextEndpointImmediate` functions.

```
OSStatus USBGetFullConfigurationDescriptor(USBPB *pb)
```

Required fields in the `USBPB` parameter block for the `USBGetFullConfigurationDescriptor` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbFlags`	Should be set to 0
`--> usbReference`	Device reference
`--> usbWValue`	Configuration index
`<-- usbBuffer`	Points to a configuration descriptor structure
`<-- usbActCount`	Size of descriptor returned

The `USBGetFullConfigurationDescriptor` function determines the size of a full configuration descriptor, including all interface and endpoint descriptors for a given configuration, allocates memory for the configuration descriptor, and reads all the descriptors in.

You don't pass the `USBGetFullConfigurationDescriptor` function a buffer pointer, the function allocates one and passes a pointer back in the `usbBuffer` field of the parameter block. The memory for the configuration descriptor must be deallocated when the information is no longer needed. The `USBDeallocMem` function should be used in the class driver's finalize routine for deallocating memory and disposing of the descriptor.

The `USBGetFullConfigurationDescriptor` function is unusual in that it takes a configuration index in the `usbWValue` field rather than a configuration value. The configuration value is found in the configuration descriptor, and is not available until the descriptor has been read. The configuration index refers to the 1st, 2nd, 3rd, or greater configuration descriptor in a device by specifying 0, 1, 2, or greater respectively. The configuration index is independent of the configuration value found in the configuration descriptor. The configuration value is used as an input parameter to set the configuration for a device.

Currently there are no other functions in the USB configuration services that provide the same functionality as the `USBGetFullConfigurationDescriptor` function. Configuration descriptors can be retrieved using the `USBGetConfigurationDescriptor` function, but the driver has to find the length of the configuration descriptor and allocate the memory for the descriptor when calling the function. Specific types of descriptors can be found with the `USBFindNextAssociatedDescriptor` function.

Once you have obtained the configuration descriptor, you need to find the interface you're interested in within the configuration descriptor by using the `USBFindNextInterfaceDescriptorImmediate` function.

USBFindNextInterfaceDescriptorImmediate

The `USBFindNextInterfaceDescriptorImmediate` function returns the address to the next interface descriptor in a specified configuration descriptor.

```
OSStatus USBFindNextInterfaceDescriptorImmediate(USBPB *pb)
```

Required fields in the `USBPB` parameter block for the `USBFindNextInterfaceDescriptorImmediate` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbBuffer`	--> Configuration descriptor <-- Interface descriptor
`--> usbFlags`	Should be set to 0
`<-- usbActcount`	Length of interface descriptor found
`<--> usbReqCount`	--> 0, This should be set to 0 the first time the call is made. Otherwise, the value from the last call should be left alone. <-- Offset of this descriptor from the start of the configuration descriptor
`<--> usbClassType`	--> Class; 0 matches any class <-- Class value for interface found
`<--> usbSubclass`	--> Subclass; 0 matches any subclass <-- Subclass value for interface found

`<--> usbProtocol`	--> Protocol; 0 matches any protocol <-- Protocol value for interface found
`<--> usbWValue`	--> 0 Configuration number: If more than one interface is described in the configuration descriptor, this field specifies the absolute number of the interface in the list.
`<-- usbWIndex`	Interface number
`<-- usbOther`	Alternate interface

The `usbReqCount` field should be set to 0 for the first iteration of this call. For each subsequent call to the `USBFindNextInterfaceDescriptorImmediate` function, `usbReqCount` contains the offset of the current interface descriptor from the beginning of the configuration descriptor.

The `usbBuffer` field should be assigned the address of the start of the configuration descriptor obtained from a call to the `USBGetFullConfigurationDescriptor` function. This must be the full configuration descriptor returned by `USBGetFullConfigurationDescriptor`. The `usbBuffer` is assigned a pointer to the next interface descriptor within the specified configuration for each subsequent call to the `USBFindNextInterfaceDescriptorImmediate` function.

The `usbClass`, `usbSubclass`, and `usbProtocol` fields should contain either specific class, subclass, and protocol numbers, or contain 0 to use for a wildcard search if the caller wants to find an interface regardless of these fields. Upon return, these fields contain the class, subclass, and protocol values for the next interface found. If the caller wants to perform a wildcard search again, the wildcard values must be reset, because these fields are filled in with the returned values from the last call.

Once you've found an interface in the device, you need to find the endpoints that make up that interface.

If no interface is found that matches the requested interface, `kUSBNotFound` is returned.

The errors returned by the `USBFindNextInterfaceDescriptorImmediate` function include:

`kUSBNotFound`	interface specified is not in configuration
`kUSBInternalErr`, `paramErr`	not a valid configuration descriptor

USBFindNextEndpointDescriptorImmediate

The `USBFindNextEndpointDescriptorImmediate` function returns the address to the next endpoint descriptor in a configuration descriptor that follows a specified interface descriptor. This is a synchronous call.

```
OSStatus USBFindNextEndpointDescriptorImmediate(USBPB *pb)
```

Required fields in the `USBPB` parameter block for the `USBFindNextEndpointDescriptorImmediate` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbFlags`	--> Direction of endpoint (`kUSBIn`, `kUSBOut`, or `kUSBAnyDirn`) <-- Direction is returned here if `kUSBAnyDirn` is used in the `usbClassType` field. Note that if `kUSBAnyDirn` is specified, this field is altered on the calls return. If you want to make another call to find an endpoint of any direction, `kUSBAnyDirn` must be specified again. Direction is also returned if `kUSBIn` or `kUSBOut` are specified. It will however, be the same value as that passed in.
`<--> usbBuffer`	--> Interface descriptor on the first call, points to an endpoint descriptor on subsequent calls <-- Endpoint descriptor
`<--> usbReqCount`	Offset of interface or endpoint descriptor in configuration descriptor
`<-- usbActcount`	Length of endpoint descriptor found
`<--> usbClassType`	--> Specific endpoint type, or `kUSBAnyType` as wildcard <-- Endpoint type
`<--> usbOther`	--> Endpoint number, always pass 0 unless you want to match a specific endpoint number. <-- Next matching endpoint is returned
`<-- usbWValue`	Maximum packet size of endpoint

The `usbBuffer` should be assigned the address of the start of the interface descriptor obtained from a call to the `USBFindNextInterfaceDescriptorImmediate`

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

function. For each subsequent call to `USBFindNextEndpointDescriptorImmediate`, `usbBuffer` is assigned a pointer to the next endpoint descriptor within the specified interface.

The errors returned by the `USBFindNextEndpointDescriptorImmediate` function include:

`kUSBNotFound`	-6987	endpoint specified is not in configuration
`kUSBInternalErr, paramErr`	-6999	not a valid configuration descriptor

Opening a Pipe

In order to communicate with an endpoint you must first open a pipe to an individual endpoint. The `USBOpenPipe` function provides a mechanism for opening a pipe.

Note
Normally the open pipe operation would be performed with the `USBConfigureInterface` function and the pipe discovered with the `USBFindNextPipe` function. However, if you are reading descriptors yourself, you may need to use the `USBOpenPipe` function. This method of opening a pipe is discouraged for future compatibility.

IMPORTANT
The `USBOpenPipe` and `USBClosePipe` functions should only be used in conjunction with the functions described in “USB Descriptor Functions,” not in conjunction with other configuration services.

USBOpenPipe

The `USBOpenPipe` function verifies that a specified device has the specified endpoint and then sets up a connection to that endpoint. It also determines the type of endpoint (control, isochronous, interrupt, or bulk). A reference (`USBPipeRef`) to this connection is returned. This reference is passed to the USB on all subsequent data transactions for this pipe. See the "USB Transaction Functions" (page 4-65) for information about performing data transactions.

```
OSStatus USBOpenPipe(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBOpenPipe` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbReference`	--> Device reference of device to be opened <-- Reference to the pipe opened
`--> usbClassType`	Type of endpoint (`kUSBBulk`, `kUSBInterrupt`, `kUSBIsoc`)
`--> usbOther`	The endpoint number within the device to open a pipe to, the endpoint number from the endpoint descriptor
`--> usbWValue`	`maxPacketSize`
`--> usbFlags`	Indicates whether the direction of the endpoint to be opened is in or out (`kUSBIn`, `kUSBOut`)

Note
At present there is little error checking performed by this function.

The errors returned by the `USBOpenPipe` function include:

`kUSBUnknownDeviceErr`	-6998	`usbReference` does not refer to a current device
`kUSBIncorrectTypeErr`	-6995	tried to open a control or isochronous pipe
`kUSBTooManyPipesErr`	-6996	ran out of internal structures

USBClosePipeByReference

The `USBClosePipeByReference` function closes the specified pipe currently opened. The pipe reference is deleted and no further reference can be made to

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

it. All outstanding transactions on the specified pipe are returned with a `kUSBAborted` status.

```
OSStatus USBClosePipeByReference(USBPipeRef ref);
```

`--> ref`	Pipe reference for the pipe to close

The `USBClosePipeByReference` function is similar to the pipe state control functions described later in this document. This function does not require a parameter block and does not complete asynchronously.

See also the `USBFindNextPipe` and `USBOpenPipe` functions.

The errors returned by the `USBClosePipeByReference` function include:

`kUSBUnknownPipeErr`	-6997	pipe reference not recognized

The `kUSBPipeStalledErr` and `kUSBPipeIdleErr` are also returned if the pipe is not currently active. In this instance, the call has failed and the pipe will have to be activated again before it can be closed. This behavior will be changed to return noErr and succeed in a later release of the USB software.

`kUSBPipeStalledErr`	-6979	pipe is stalled, pipe not closed
`kUSBPipeIdleErr`	-6980	pipe is idle, pipe not closed

USBGetConfigurationDescriptor

The `USBGetConfigurationDescriptor` function gives class drivers access to the USB configuration descriptor.

The `USBGetConfigurationDescriptor` function returns configuration descriptors that define the contents of the configuration data for the device. The configuration descriptor is 9 bytes, and is followed by all the interface descriptors complete with their associated endpoint descriptors as well as any class or vendor specific descriptors. The `USBGetConfigurationDescriptor` function returns as much of this data for one configuration as requested.

```
OSStatus USBGetConfigurationDescriptor(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBGetConfigurationDescriptor` function are

`--> pbLength`	Length of parameter block

-->	`pbVersion`	Parameter block version number
-->	`usbCompletion`	The completion routine
-->	`usbRefcon`	General-purpose value passed back to the completion routine
-->	`usbReference`	Device reference
-->	`usbWValue`	Configuration number
-->	`usbReqCount`	Amount of configuration data requested
-->	`usbBuffer`	--> Pointer to the address to store the data in
<--	`usbActCount`	Actual amount of data returned
-->	`usbFlags`	Should be set to 0

The `USBGetConfigurationDescriptor` function differs from the `USBGetFullConfigurationDescriptor` function in that it allows the calling driver to specify how much configuration data the function should return. `USBGetConfigurationDescriptor` allows the caller to get either the 9-byte configuration descriptor (`USBConfigurationDescriptor`), a descriptor specified in `usbWValue`, or as much of the configuration data as requested.

The `USBGetConfigurationDescriptor` function requires the caller to allocate the memory for the returned data and pass a pointer to the address of the allocated memory block in `usbBuffer`. The caller must also specify how many bytes of the configuration data to return to the buffer in the `usbReqCount` field.

The `usbReqCount` field specifies the largest amount of data that you want returned. If the descriptor has less data, less data is returned. If the descriptor has more data, only the requested amount of data is returned. This is not an error condition.

Device Management Functions

The functions here provide a mechanism for the hub driver to inform the USB Manager of the addition of a new device to a specified USB and to retrieve the USB device ID of the new USB device. The USB hub driver tasks include assigning the device address and preparing the device's default pipe for use. This is the process of enumerating a device for use on the USB.

The functions in this section are called in a specific sequence to perform proper device enumeration on the USB. The function calling sequence is as follows:

1. Call the `USBHubAddDevice` function

2. When `USBHubAddDevice` completes, the hub driver is clear to reset the device.
3. Reset the port for that device
4. Obtain the port status from the hub to get the speed of the device.
5. Call `USBHubConfigurePipeZero` to set the device speed and packet size.
6. Perform any necessary communication with the device.
7. Call the `USBHubSetAddress` function to set the address of the device which is used for future communication by the hub driver and other drivers interested in the device.

USBHubAddDevice

The `USBHubAddDevice` function informs the USB that the hub driver has a new device that needs to be added to the USB.

At the time the device first appears on the USB, it has not yet been reset. When the completion routine for this function is called, it is safe for the hub driver to reset the device. The hub driver should then read the device descriptor to get the endpoint and maxpacket size. The driver must then call the `USBHubSetAddress` function, even if an error occurs. There is a 1 second timeout allowed, if `USBHubSetAddress` is not called, it will be called for you. This is done to prevent the bus enumeration mechanism from halting further system activity.

```
OSStatus USBHubAddDevice(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBHubAddDevice` function are

-->	`pbLength`	Length of parameter block
-->	`pbVersion`	Parameter block version number
-->	`usbCompletion`	The completion routine
-->	`usbRefcon`	General-purpose value passed back to the completion routine
<--	`usbReference`	Device reference

The device reference can be used to address the device while it is device 0. This reference becomes invalid as soon as it is addressed with the `USBHudSetAddress` function.

There must be a valid `USBReference` passed in the `usbReference` field of the parameter block. If a `nil` value or a reference that does not match an existing device, interface, or pipe is passed in, the call returns immediately with an unknown device error.

If the device associated with this call is unplugged and its driver removed while this function call is pending, the function will not complete.

USBHubConfigurePipeZero

The `USBHubConfigurePipeZero` function must be called after a device is reset, and before any attempt is made to communicate with the device at its default address zero.

```
OSStatus USBHubConfigurePipeZero(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBHubConfigurePipeZero` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Device zero reference returned from the USBHubAddDevice function.
`--> usbFlags`	Device speed 0 or 1: slow (1 indicates a low speed device)
`--> usbWValue`	maxPacketSize, obtained from the device descriptor

USBHubSetAddress

The `USBHubSetAddress` function addresses the currently unaddressed device (The device has been reset and is responding as device 0.) and creates a device reference for it. After this function completes, the device can be addressed with device requests using the new device reference.

```
OSStatus USBHubSetAddress(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBHubSetAddress` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`<--> usbReference`	--> Reference from USBHubAddDevice call <-- New device reference
`--> usbFlags`	0 or 1: (1 indicates a low speed device)
`--> usbWValue`	`maxPacketSize` of endpoint 0, obtained from the device descriptor

USBHubDeviceRemoved

The `USBHubDeviceRemoved` function causes all pipes open to the specified device to be closed, thus removing the device.

```
OSStatus USBHubDeviceRemoved(USBPB *pb);
```

Required fields in the `USBPB` parameter block for the `USBHubDeviceRemoved` function are

`--> pbLength`	Length of parameter block
`--> pbVersion`	Parameter block version number
`--> usbCompletion`	The completion routine
`--> usbRefcon`	General-purpose value passed back to the completion routine
`--> usbReference`	Device reference
`--> usbFlags`	Should be set to 0

The hub driver should also call the `USBExpertRemoveDeviceDriver` function (page 4-77) to inform the USB Manager. This removes the class driver associated with the device. If the device is a hub, the hub driver should make the device removal calls for all devices attached to the hub. The USB Manager is responsible for determining what if any downstream devices are connected to the hub and disconnecting them to remove that hub's tree of devices.

Constants and Data Structures

This section lists the constants and data structures used by the USL. Always check the USB.h header file for the current version of the constants and structures that support Mac OS USB driver development.

USB Constants

The constants recognized by the USL are listed in this section.

Parameter Block Constants

```
kUSBCurrentPBVersion    = 0x0100        /* version 1.00*/
kUSBIsocPBVersion       = 0x0109        /* version 1.10*/
```

Endpoint Type Constants

```
kUSBControl     = 0
kUSBIsoc        = 1
kUSBBulk        = 2
kUSBInterrupt   = 3
kUSBAnyType     = 0xff
```

usbBMRequest Direction Constants

```
kUSBOut         = 0
kUSBIn          = 1
kUSBNone        = 2
kUSBAnyDirn     = 3
```

usbBMRequestType Type Constants

```
kUSBStandard    = 0
kUSBClass       = 1
kUSBVendor      = 2
```

usbBMRequest Recipient Constants

```
kUSBDevice      = 0
kUSBInterface   = 1
kUSBEndpoint    = 2
kUSBOther       = 3
```

usbBRequest Constants

```
kUSBRqGetStatus     = 0
kUSBRqClearFeature  = 1
kUSBRqReserved1     = 2
kUSBRqSetFeature    = 3
kUSBRqReserved2     = 4
kUSBRqSetAddress    = 5
kUSBRqGetDescriptor = 6
kUSBRqSetDescriptor = 7
kUSBRqGetConfig     = 8
kUSBRqSetConfig     = 9
kUSBRqGetInterface  = 10
kUSBRqSetInterface  = 11
kUSBRqSyncFrame     = 12
```

Interface Constants

```
    kUSBHIDInterfaceClass          = 0x03
    kUSBNoInterfaceSubClass        = 0x00
    kUSBBootInterfaceSubClass      = 0x01
```

Interface Protocol Constants

```
    kUSBNoInterfaceProtocol        = 0x00
    kUSBKeyboardInterfaceProtocol  = 0x01
    kUSBMouseInterfaceProtocol     = 0x02
```

Driver Class Constants

```
    kUSBCompositeClass        = 0,
    kUSBAudioClass            = 1
    kUSBCOMMClass             = 2,
    kUSBHIDClass              = 3,
    kUSBDisplayClass          = 4,
    kUSBPrintingClass         = 7
    kUSBMassStorageClass      = 8
    kUSBHubClass              = 9,
    kUSBDataClass             = 10
    kUSBVenderSpecificClass   = 0xFF
};
```

Descriptor Type Constants

```
    kUSBDeviceDesc      = 1
    kUSBConfDesc        = 2
    kUSBStringDesc      = 3
    kUSBInterfaceDesc   = 4
    kUSBEndpointDesc    = 5
    kUSBHIDDesc         = 0x21
    kUSBReportDesc      = 0x22
    kUSBPhysicalDesc    = 0x23
    kUSBHUBDesc         = 0x29
```

Pipe State Constants

```
kUSBActive      = 0,    /* Pipe can accept new transactions*/
kUSBIdle        = 1,    /* Pipe cannot accept new transactions*/
kUSBStalled     = 2     /* An error occured on the pipe*/
```

USB Power and Bus Attribute Constants

```
kUSB100mAAvailable  = 50
kUSB500mAAvailable  = 250
kUSB100mA           = 50
kUSBAtrBusPowered   = 0x80
kUSBAtrSelfPowered  = 0x40
kUSBAtrRemoteWakeup = 0x20
```

Driver File and Resource Types

```
kServiceCategoryUSB = FOUR_CHAR_CODE('usb ')
kUSBTypeIsHub       = FOUR_CHAR_CODE('hubd')
kUSBTypeIsHID       = FOUR_CHAR_CODE('HIDd')
kUSBTypeIsDisplay   = FOUR_CHAR_CODE('disp')
kUSBTypeIsModem     = FOUR_CHAR_CODE('modm')
kUSBDriverFileType  = FOUR_CHAR_CODE('ndrv')
kUSBDriverRsrcType  = FOUR_CHAR_CODE('usbd')
kUSBShimRsrcType    = FOUR_CHAR_CODE('usbs')
kTheUSBDriverDescriptionSignature = FOUR_CHAR_CODE('usbd')
```

USB Data Structures

These are the data structures defined by the USL for USB device descriptors. The current definitions can also be found in the USB.h file.

Driver Plug-in Dispatch Table Structure

The driver dispatch table used to match and initialize the appropriate driver with a device is of the form shown here. This structure is filled in by the class driver.

```
struct USBClassDriverPluginDispatchTable {
    UInt32                         pluginVersion;
                                   /* Version number of this */
                                   /* plugin for the dispatch table */
    USBDValidateHWProcPtr          validateHWProc;
                                   /* Pointer to */
                                   /* the procedure the driver */
                                   /* uses to verify that device is */
                                   /* the proper hardware */
    USBDInitializeDeviceProcPtr    initializeDeviceProc;
                                   /* Pointer to */
                                   /* the procedure that initializes */
                                   /* the class driver */
    USBDInitializeInterfaceProcPtr initializeInterfaceProc;
                                   /* Pointer to procedure that */
                                   /* initializes a particular */
                                   /* interface in the class driver.*/
    USBDFinalizeProcPtr            finalizeProc;
                                   /* Pointer to the procedure that */
                                   /* finalizes the class driver */
    USBDDriverNotifyProcPtr        notificationProc;
                                   /* Pointer to the procedure that */
                                   /* passes notifications to */
                                   /* the driver */
```

Device Descriptor Structure

The USB device descriptor is of this form:

```
struct USBDeviceDescriptor {
    UInt8       length;
    UInt8       descType;
    UInt16      usbRel;
    UInt8       deviceClass;
    UInt8       deviceSubClass;
    UInt8       protocol;
    UInt8       maxPacketSize;
    UInt16      vendor;
    UInt16      product;
    UInt16      devRel;
    UInt8       manuIdx;
    UInt8       prodIdx;
    UInt8       serialIdx;
    UInt8       numConf;
    UInt16      descEnd;
};
```

Configuration Descriptor Structure

The USB device configuration descriptor is of this form:

```
struct USBConfigurationDescriptor {
    UInt8       length;
    UInt8       descriptorType;
    UInt16      totalLength;
    UInt8       numInterfaces;
    UInt8       configValue;
    UInt8       configStrIndex;
    UInt8       attributes;
    UInt8       maxPower;
};
```

Interface Descriptor Structure

The USB device interface descriptor is of this form:

```
struct USBInterfaceDescriptor {
    UInt8       length;
    UInt8       descriptorType;
    UInt8       interfaceNumber;
    UInt8       alternateSetting;
    UInt8       numEndpoints;
    UInt8       interfaceClass;
    UInt8       interfaceSubClass;
    UInt8       interfaceProtocol;
    UInt8       interfaceStrIndex;
};
```

Endpoint Descriptor Structure

The USB device endpoint descriptor is of this form:

```
struct USBEndPointDescriptor {
    UInt8       length;
    UInt8       descriptorType;
    UInt8       endpointAddress;
    UInt8       attributes;
    UInt16      maxPacketSize;
    UInt8       interval;
};
```

HID Descriptor Structure

The USB HID descriptor is of this form:

```
struct USBHIDDescriptor {
    UInt8       descLen;
    UInt8       descType;
    UInt16      descVersNum;
    UInt8       hidCountryCode;
    UInt8       hidNumDescriptors;
    UInt8       hidDescriptorType;
    UInt8       hidDescriptorLengthLo;
    UInt8       hidDescriptorLengthHi;
};
```

HID Report Descriptor Structure

The USB HID report descriptor is of this form:

```
struct USBHIDReportDesc {
    UInt8       hidDescriptorType;
    UInt8       hidDescriptorLengthLo;
    UInt8       hidDescriptorLengthHi;
};
```

Hub Port Status Structure

The hub port status structure is of this form:

```
struct USBHubPortStatus {
    UInt16      portFlags;          /* Port status flags */
    UInt16      portChangeFlags;    /* Port changed flags */
};
```

USL Error Codes

Error codes returned by the USL are in the range -6900 to -6999 as listed in Table 4-2.

Table 4-2 Error definitions

Error constant	Number	Definition
`kUSBNoErr`	0	No error occurred
`kUSBInternalErr`	-6999	Internal error
`kUSBUnknownDeviceErr`	-6998	Device reference not recognized
`kUSBUnknownPipeErr`	-6997	Pipe reference not recognized
`kUSBTooManyPipesErr`	-6996	Too many pipes
`kUSBIncorrectTypeErr`	-6995	Incorrect type specified
`kUSBRqErr`	-6994	Request error
`kUSBUnknownRequestErr`	-6993	Unknown request
`kUSBTooManyTransactionsErr`	-6992	Too many transactions
`kUSBAlreadyOpenErr`	-6991	Device already open
`kUSBNoDeviceErr`	-6990	No device
`kUSBDeviceErr`	-6989	Device error
`kUSBOutOfMemoryErr`	-6988	Out of memory
`kUSBNotFound`	-6987	USB not found
`kUSBPBVersionError`	-6986	Wrong parameter block version
`kUSBPBLengthError`	-6985	pbLength too small

Table 4-2 Error definitions

Error constant	Number	Definition
`kUSBCompletionError`	-6984	No completion routine specified
`kUSBFlagsError`	-6983	Flags not initialized to 0
`kUSBAbortedError`	-6982	Pipe aborted
`kUSBNoBandwidthError`	-6981	Not enough bandwidth available
`kUSBPipeIdleError`	-6980	Pipe is idle; it cannot accept transactions
`kUSBPipeStalledError`	-6979	Pipe has stalled; it cannot be used until the error is cleared with a `USBClearPipeStallByReference` call
`kUSBUnknownInterfaceErr`	-6978	Interface reference not recognized
`kUSBDeviceBusy`	-6977	Device is already being configured
`kUSBDevicePowerProblem`	-6976	Device has a power problem
`kUSBLinkErr`	-6916	Link error
`kUSBCRCErr`	-6915	Pipe stall: bad CRC
`kUSBBitstufErr`	-6914	Pipe stall: bitstuffing
`kUSBDataToggleErr`	-6913	Pipe stall: bad data toggle
`kUSBEndpointStallErr`	-6912	Device didn't understand
`kUSBNotRespondingErr`	-6911	Pipe stall, no device, or device hung
`kUSBPIDCheckErr`	-6910	Pipe stall: PID CRC error
`kUSBWrongPIDErr`	-6909	Pipe stall: Bad or wrong PID
`kUSBOverRunErr`	-6908	Packet too large or more data than buffer
`kUSBUnderRunErr`	-6907	Less data than buffer
`kUSBBufOvrRunErr`	-6904	Host hardware failure on data in

Table 4-2 Error definitions

Error constant	**Number**	**Definition**
`kUSBBufUnderRunErr`	`-6903`	Host hardware failure on data out
`kUSBNotSent1Err`	`-6902`	Transaction not sent
`kUSBNotSent2Err`	`-6901`	Transaction not sent

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

CHAPTER 5

USB Manager Reference

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

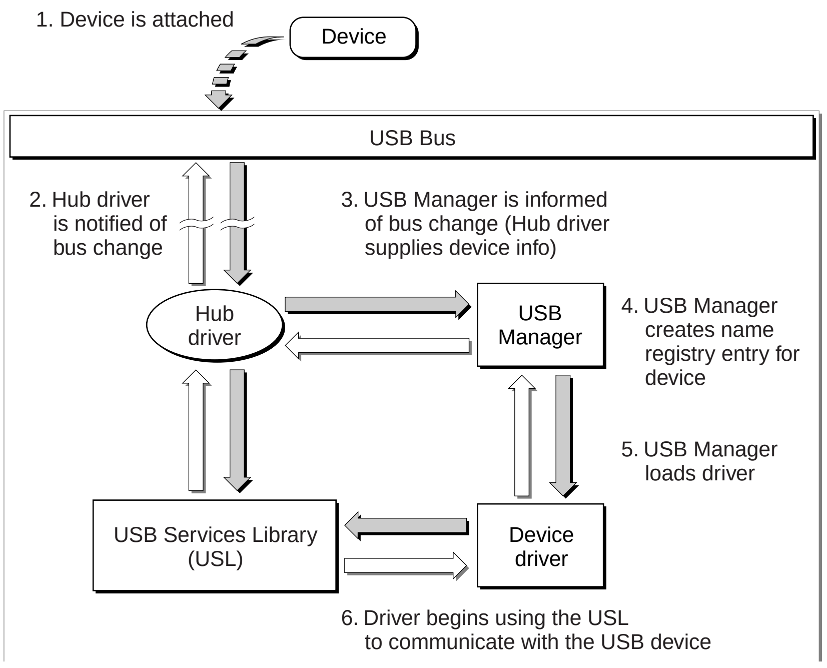
The USB Manager API is described in this chapter.

Overview

The USB Manager maintains a database of all the currently connected devices that communicate using the USB protocol. Whenever a device is added to the USB, it is the responsibility of the USB Manager to register the device with the Name Registry and load the device's driver software. In the event of a device being removed, the USB Manager must ensure that the driver is removed cleanly from the system and all references to the device in the Name Registry are removed.

Figure 5-1 depicts the sequence of events that the USB Manager participates in when a device is added to the USB.

Figure 5-1 Device addition event sequence on the USB


1. Device is attached
Device
USB Bus
2. Hub driver is notified of bus change
3. USB Manager is informed of bus change (Hub driver supplies device info)
Hub driver
USB Manager
4. USB Manager creates name registry entry for device
5. USB Manager loads driver
USB Services Library (USL)
Device driver
6. Driver begins using the USL to communicate with the USB device

The USB Manager consists of native code fragments wrapped in a file of type `'expt'`. During the Macintosh boot sequence, the USB Manager is loaded immediately following all native drivers ('ndrv') and before generic INIT files. The USB Manager resides in the Extensions folder.

The USB Manager is responsible for the following services which support the USB architecture.

- Maintain USB topology in database: keep updated information about the USB in the Name Registry and dynamically update the information as devices are added or removed from the bus.
- Provide access functions for database information: Device information needed by either the USL or a device driver should be accessible via the USB Manager.
- Generate unique opaque bus reference, a `USBBusRef` defined is a USBReference (SInt32), when a root hub is detected/loaded. For possible future use, a unique bus reference is generated by the USB Manager for each instantiated root hub. Every device record stores the bus reference of the bus.

USB Manager API

This section describes the data structures and functions supported by the USB Manager API. In this chapter functions refers to the function declarations for the APIs rather than functions within USB devices.

Prototypes for all functions and definitions of other related data types are in the USB.h header file. The file is typically found in the includes folder.

Topology Database Access Functions

The functions for getting information about the USB topology are defined in this section.

Getting Device Descriptors

The `USBGetDeviceDescriptor` function returns a pointer to the device descriptor of the specified device reference.

```
OSStatus USBGetDeviceDescriptor (
                    USBDeviceRef *deviceRef,
                    USBDeviceDescriptor *deviceDescriptor,
                    UInt32 size);
```

`--> deviceRef`	A pointer to the allocated device reference for which you want the device descriptor.
`<-- deviceDescriptor`	A pointer to the device descriptor.
`--> size`	Size of the descriptor. If the descriptor that is returned is larger than the requested size, a `kUSBOverRunErr` is returned and only the first size bytes of the descriptor are filled in.

Getting Interface Descriptors

The `USBGetInterfaceDescriptor` function returns a pointer to the interface descriptor of supplied device reference.

```
OSStatus USBGetInterfaceDescriptor (
                    USBInterfaceRef *interfaceRef,
                    USBInterfaceDescriptor *InterfaceDescriptor,
                    UInt32 size);
```

`--> interfaceRef`	A pointer to the allocated interface reference for which you want the interface descriptor.
`<-- interfaceDescriptor`	A pointer to the device interface descriptor.
`--> size`	Size of the descriptor. If the descriptor that is returned is larger than the requested size, a `kUSBOverRunErr` is returned and only the first size bytes of the descriptor are filled in.

Finding The Driver For A Device By Class

The USBGetNextDeviceByClass function returns a class driver reference for the class driver matching the specified device class and optionally the device subclass for that device. This function also works with interface references.

```
OSStatus USBGetNextDeviceByClass (
                                USBDeviceRef *deviceRef,
                                CFragConnectionID *connID,
                                UInt16 theClass,
                                UInt16 theSubClass,
                                UInt16 theProtocol);
```

`<--> deviceRef`	A pointer to the device or interface driver reference for the device or interface class specified.
`<-- connID`	A pointer to the device connection ID.
`--> theClass`	A number representing the device or interface class for which you want a compatible class driver. You can pass in kUSBAnyClass as a wildcard value. See the USB Specification for the device and interface class descriptions and identifiers.
`--> theSubClass`	A number representing the device or interface sub class for which you want a compatible class driver. You can pass in kUSBAnySubClass as a wildcard value. See the USB Specification for the device and interface subclass descriptions and identifiers.
`--> theProtocol`	A number representing the device or interface protocol for which you want a compatible class driver. You can pass in kUSBAnyProtocol as a wildcard value. See the USB Specification for the device and interface protocol descriptions and identifiers.

The USBGetNextDeviceByClass function returns a pointer to the next usbDeviceRef for a class driver matching the specified deviceClass and (optionally) deviceSubClass and deviceProtocol parameters. Pass kNoDeviceRef for the deviceRef parameter to begin, then pass the returned device reference for subsequent searches.

An OSStatus error of -43 is returned if a device cannot be found with the specified parameters. The device reference, deviceRef, returns unchanged if no subsequent match is made. The typical way to find all similar devices is to keep calling the `USBGetNextDeviceByClass` function until the status value changes from `noErr`. At that point, the `deviceRef` is officially undefined.

The driver descriptor structure must have the same class and subclass codes as the codes for the device that is specified in the function call. This is particularly important for vendor specific devices, since the correct driver for the device would not typically load if the class and subclass codes don't match those for the device.

If you are developing a device and the `USBGetNextDeviceByClass` function isn't finding the requested device, be sure that the driver descriptor structure for your device driver has the same class and subclass codes as the device.

Constants are defined for the device class, subclass, protocol, vendor, and product identifiers which you can pass as wildcard values in the functions `USBGetNextDeviceByClass` and `USBInstallDeviceNotification` (page 5-115).

Constant	Value	Description
`kUSBAnyClass`	`0xffff`	Pass in as a wildcard in the `deviceClass` parameter or `usbClass` field in the device notification parameter block.
`kUSBAnySubClass`	`0xffff`	Pass in as a wildcard in the `deviceSubClass` parameter or `usbSubClass` field in the device notification parameter block
`kUSBAnyProtocol`	`0xffff`	Pass in as a wildcard in the `deviceProtocol` parameter or `usbProtocol` field in the device notification parameter block
`kUSBAnyVendor`	`0xffff`	Pass in as a wildcard in the `usbVendor` field in the device notification parameter block.
`kUSBAnyClass`	`0xffff`	Pass in as a wildcard in the `usbProduct` field in the device notification parameter block

Note
In USB version 1.0.1 (the iMac update 1.0) a bug prevented correct searches if `usbClass`, `usbSubclass`, and `usbProtocol` were equal 0 and `kNoDeviceRef` is used for the `deviceRef`. This behavior is not present in version 1.1 and greater of the Mac OS USB software.

Getting The Connection ID For Class Driver

The `USBGetDriverConnectionID` function returns a pointer to the `CFragConnectionID` of the driver referenced by the device reference.

```
OSStatus USBGetDriverConnectionID (
                    USBDeviceRef *deviceRef,
                    CFragConnectionID *connID);
```

`--> deviceRef`	A pointer to the device reference for which you want the connection ID.
`<-- connID`	A pointer to the connection ID.

Getting The Bus Reference For a Device

The `USBDeviceRefToBusRef` function returns a pointer to the bus reference for the device specified with a device reference.

```
OSStatus USBDeviceRefToBusRef (
                    USBDeviceRef *deviceRef,
                    USBBusRef *busRef);
```

`--> deviceRef`	A pointer to an already established device reference for which you want the bus reference.
`<-- busRef`	A pointer to the bus reference.

Callback Routine for Device Notification

The callback routine, callback routine parameter block, and callback notification request functions used for device notification are listed in this section.

The device notification mechanism is used to inform clients when devices are added and removed from the USB. Clients register for notification services using the `USBInstallDeviceNotification` function and can request all notifications or a specific notification type. Whenever a device or interface is added or removed from the bus, all registered clients are called back with the information about the device or interface.

Clients that register for notifications must be sure to un-register with the `USBRemoveDeviceNotification` function before their code fragment is unloaded.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

The callback routine is always called at task time, and may allocate memory, make toolbox calls, or perform other system maintenance operations.

Device Notification Callback Routine

The device notification callback routine declaration is defined as:

```
typedef void (USBDeviceNotificationCallbackProc)
                    (USBDeviceNotificationParameterBlockPtr pb);

typedef USBDeviceNotificationCallbackProc
*USBDeviceNotificationCallbackProcPtr;
```

Device Notification Parameter Block

The parameter block for the device notification callback routine is defined as:

```
/* Device Notification Parameter Block */
struct USBDeviceNotificationParameterBlock
{
UInt16                                  pbLength;
UInt16                                  pbVersion;
USBNotificationType                     usbDeviceNotification;
UInt8                                   reserved1;
USBDeviceRef                            usbDeviceRef;
UInt16                                  usbClass;
UInt16                                  usbSubClass;
UInt16                                  usbProtocol;
UInt16                                  usbVendor;
UInt16                                  usbProduct;
OSStatus                                result;
UInt32                                  token;
USBDeviceNotificationCallbackProcPtr    callback;
UInt32                                  refcon;
};
```

Field descriptions

`<--> usbDeviceNotification`	The type of notification
	The following notifications are defined:
	`kNotifyAnyEvent`
	`kNotifyAddDevice`

	`kNotifyAddInterface` `kNotifyRemoveDevice` `kNotifyRemoveInterface`
`<-- usbDeviceRef`	The device reference for the target device
`<--> usbClass`	The class of the target device, use `kUSBAnyClass` for any class
`<--> usbSubClass`	The subclass of the target device, use `kUSBAnySubClass` for any subclass
`<--> usbProtocol`	The protocol of the target device, use `kUSBAnyProtocol` for any protocol
`<--> usbVendor`	The vendor ID of the target device, use `kUSBAnyVendor` for any vendor
`<--> usbProduct`	The product ID of the target device, use `kUSBAnyProduct` for any product
`<-- result`	The status of the call
`<-- token`	The identifier for this notification request
`--> callback`	A pointer to the callback routine to be called when the notification criteria is satisfied

Installing The Device Callback Request

The `USBInstallDeviceNotification` function installs the device notification routine for the device specified in the `USBDeviceNotificationParameterBlock`. Pass in `0xffff` or the wildcard constants as a wildcard for class, subclass, protocol, vendor, and/or product. Pass in `kNotifyAnyEvent` (0xff) in the `usbDeviceNotification` field to be notified for any change that occurs.

```
void USBInstallDeviceNotification(
                    USBDeviceNotificationParameterBlock *pb);
```

`pb`	A pointer to the `USBDeviceNotificationParameterBlock` defined on (page 5-114).

If a code fragment installs a device notification routine, the device notification routine must be removed with the `USBRemoveDeviceNotification` function before the code fragment is unloaded.

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Removing The Device Callback Request

The `USBRemoveDeviceNotification` function removes a previously installed device notification routine.

```
OSStatus USBRemoveDeviceNotification (UInt32 token);
```

`token`	Notification identifier from the previously installed device notification routine.

Changes In Mac OS USB Version 1.1

A general discussion of the features in version 1.1 of the Mac OS USB software is provided in this appendix. It also lists and defines the use of the version 1.1 parameter block that supports isochronous transfers.

There are significant differences in the features supported in version 1.1 of the Mac OS USB software. To take advantage of the new features some modification of existing code is required. For information about the required code changes to support version 1.1, see "Code Changes Required To Support The Version 1.1 USBPB" (page 123).

Major Feature Updates In Version 1.1

The major feature enhancements included in version 1.1 of the Mac OS USB software are:

- Isochronous support, new parameter block (page 119)
- Multiple bus support
- Improved bus enumeration
- Driver notification messages that support Mac OS sleep and wake
- Improved functionality for USB control requests

IMPORTANT

It should be noted that although the features listed here are supported by the version 1.1 `USBPB` parameter block, all Macintosh computers that support USB may not include the necessary ROM code to implement the features. Always check the USB gestalt selectors, defined in "Isochronous Transfer Support" (page 118) and "USB Software Presence and Version Attributes" (page 36), rather than the version number to ensure that the features you are interested in are supported on the Macintosh your software is running on.

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

Improved Bus Enumeration

Version 1.1 provides improved bus emueration at startup to support proper USB driver loading before other system extensions are initialized. This is accomplished by providing task time for the USB expert loader to process all hub communications. When all hubs have reported that they have discovered their devices, and the USB system software has completed the search for USB class drivers, then the remainder of the booting process, loading extensions and launching the finder, continues.

Multiple USB Bus Support

The Mac OS USB version 1.1 software supports multiple USB buses on a system. If you are looking through the name registry, you need to check every USB controller node for attached hubs and devices.

Driver Notification Messages

Additional messages have been defined for handling Mac OS power management features. Version 1.1 of the Mac OS USB software notifies class drivers through the `USBDriverNotificationProcPtr` with the following messages:

Message constant name
`kNotifyUSBSystemSleepRequest`
`kNotifyUSBSystemSleepDemand`
`kNotifyUSBSystemSleepWakeUp`
`kNotifyUSBSystemSleepRevoke`

These messages correspond to the `Sleep` procedure selector codes defined in the Chapter 6, “Power Manager,” in Inside Macintosh, “Devices.” Your driver should return an appropriate response to these messages as defined in “Writing a Sleep Procedure” Chapter 6, “Power Manager,” Inside Macintosh: Devices.

Isochronous Transfer Support

Version 1.1 contains support for isochronous transfers. You can test for the presence of isochronous support by checking the gestalt selector `gestaltUSBAttr`

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

(‘usb ‘). If gestaltUSBHasIsoch (bit 1 = 0x02) is set, then isochronous support is available in the form of two new calls:

```
OSStatus USBIsocWrite(USBPB *pb);
OSStatus USBIsocRead(USBPB *pb);
```

Improved Functionality For USB Control Requests

A new flag was added to the USBDeviceRequest function (page 63) to allow for USB control transactions addressed to an interface or endpoint of a device. The new feature allows the call to be made without the driver explicitly knowing the number of the endpoint or interface before the call is made. The USL now fills in the interface or endpoint number when an interface or pipe reference is passed in with the call.

To use the new feature, you specifiy the flag kUSBAddessRequest in the usbFlags field of the USBDeviceRequest function. If the recipient field in BMRequestType is an endpoint or interface, the relevant endpoint or interface number is derived from the pipe or interface reference passed in the usbReference field. The appropriate interface or endpoint number is put into the usbWIndex field before the control transaction call takes place.

The Isochronous Version Of The USBPB

This section lists the isochronous version 1.1 USBPB parameter block, and discusses things to be aware of when using the isochronous variant of the USBPB.

The isochronous version 1.1 USBPB structure is defined as:

```
struct USBIsocFrame {
    OSStatus frStatus;           /* Frame status information */
    UInt16 frReqCount;           /* Bytes to transfer */
    UInt16 frActCount;           /* Actual bytes transferred */
};

struct usbIsocBits {
    USBIsocFrame *FrameList;
    UInt32 NumFrames;
};
```

```
struct usbControlBits {
    UInt8 BMRequestType;          /* For control transactions */
    UInt8 BRequest;               /* Specific control request */
    USBRqValue WValue;            /* For control transactions, the */
                                  /* value field of the setup packet */
    USBRqIndex WIndex;            /* For control transactions, the */
                                  /* value field of the setup packet */
    UInt16 reserved4;             /* Reserved */
};

struct USBPB{

    void *qlink;
    UInt16 qType;
    UInt16 pbLength;              /* Length of parameter block */
    UInt16 pbVersion;             /* Parameter block version number */
                                  /* kUSBIsocPBVersion for iscohronous */
                                  /* version 1.1 USBPB */
    UInt16 reserved1;             /* Reserved */
    UInt32 reserved2;             /* Reserved */

    OSStatus usbStatus;           /* Completion status of the call */
    USBCompletion usbCompletion;  /* Completion routine */
    UInt32 usbRefcon;             /* For use by the completion routine */
    USBReference usbReference;    /* Device, pipe, interface, endpoint */
                                  /* reference as appropriate */

    void *usbBuffer;              /* Pointer to the data to be sent */
                                  /* to or received from the device */
    USBCount usbReqCount;         /* Length of usbBuffer */
    USBCount usbActCount;         /* Number of bytes sent or received */
    USBFlags usbFlags;            /* Miscellaneous flags */

    UInt32 usbFrame;              /* Start frame of transfer */

union{
    usbControlBits cntl;           /* usbControlBits struct */
                                   /* used for control transactions */
    usbIsocBits isoc;              /* usbIsocBits frames structure */
}usb;
```

```
    UInt8 usbClassType;             /* Class for interfaces, */
                                    /* transfer type for endpoints */
    UInt8 usbSubclass;              /* Subclass for interfaces */
    UInt8 usbProtocol;              /* Protocol for interfaces */
    UInt8 usbOther;                 /* General purpose value */
    UInt32 reserved6;               /* Reserved */
    UInt16 reserved7;               /* Reserved */
    UInt16 reserved8;               /* Reserved */

    }USBPB;
```

Using the USBPB For Isochronous Transactions

This section defines how to use the fields that support isochronous transactions in the version 1.1 `USBPB`.

When making isochronous calls, you set the `pbVersion` field in the `USBPB` parameter block to `kUSBIsocPBVersion`, and use the isochronous variant of the `USBPB` parameter block.

Isochronous transfers occur on a per frame basis. Mac OS USB version 1.1 does not implement the per sample method suggested in Chapter 10 of the USB Specification 1.0. This may be added in a future release.

The isochronous transfer implementation requires managing data flow in specific frames. An isochronous pipe has a maximum number of bytes that it can transfer every frame. The pipe can transfer fewer bytes, but it can not transfer more. Each frame can generate its own error code.

To support frames for isochronous transfers, the following new structure is introduced:

```
typedef struct{
    OSStatus frStatus;
    UInt16   frReqCount
    UInt16   frActCount;
}USBIsocFrame;
```

This structure encapsualates the transfer for one frame. On entry, the value of the `frReqCount` field is set to indicate how many bytes are to be transfered (in or out) for this particular frame. (Note 16 bits is more than enough, the `maxpacketsize` is 1023 bytes, 10 bits).

On completion, the `frActCount` field indicates how many bytes were actually transfered and the frStatus field specifies the result of the attempt.

For input transfers the `frStatus` field may return errors, such as:

`kUSBUnderRunErr`	`-6907`	Packet received was shorter than expected
`kUSBOverRunErr`	`-6908`	Packet too large or more data than buffer
`kUSBCRCErr`	`-6915`	Pipe stall, bad CRC; packet was received corrupt

In all of the above cases, there is data in the `usbBuffer` (it may not be very good data) which the class driver can use as it pleases.

For output, the error code is less interesting, you only know that the packet was launched onto the bus (or not as the case may be). There is not any indication that the data packet was received correctly, or that anyone was listening to it at all.

The `usbStatus` field returns an overall status for the isochronous transaction. If usbStatus returns with no error, then the status for all of the packets is also no error. If usbStatus is returned with another status value, then all of the individual packets should be examined for error codes. The usbStatus field contains a representitive error if there are multiple packet errors.

The `usbBuffer` field points to the data, all of the packets to be sent or received are layed end to end.

The `usbReqCount` and `usbActCount` fields specify the overall total of data for all the packets sent or received.

The `usbReference` field is a pipe reference to an isochronous pipe.

The `FrameList` field (`usb.isoc.FrameList`) in the `usbIsocBits` structure is a pointer to an array of `USBIsocFrame` structures that specify the individual packets. The start of any individual packet is found by adding the values in the `frReqCount` fields for all the preceding packets and adding that to `usbBuffer`. For example, if the `frReqCount` values are (61, 62, 63, 64, and so on) and you want the address for the packet to be sent in the third frame, start with the address of `usbBuffer` and add 61 + 62.

The `NumFrames` field (`usb.isoc.NumFrames`) is the number of frames pointed to by the `FrameList` field, and also defines over how many frames the call will be active.

The `usbFrame` field specifies the frame number on which the transfers are to start. A frame is specified to be the nearest frame to the current frame with the specified low 32 bits when the transfer is called. This method eliminates the need for a 64-bit frame counter as long as the class driver has a latency of less than 23 days.

Isochronous pipes are opened when a `USBConfigureInterface` function (page 56) is called. During a call to `USBConfigureInterface` function, the available bandwidth is checked. If bandwidth is insufficient, the call to open the isochronous pipes could fail.

Code Changes Required To Support The Version 1.1 USBPB

This section describes the changes you should be aware of if you are working with code that supported the version 1.0 parameter block in USB.h and you want to take advantage of the features in version 1.1 of the Mac OS USB software.

The `USBPB` parameter block structure has been converted to include unions that provide support for isochronous transfers. The change is binary compatible (you can keep the same `kUSBCurrentVersion` value), but it is necessary to make changes to existing source code in order to use the version 1.1 USB.h file.

At the simplest level, the necessary changes can be made by doing a search and replace of the following strings in your code:

Old string	New replacement string
`usbBMRequestType`	`usb.cntl.BMRequestType`
`usbBRequest`	`usb.cntl.BRequest`
`usbWValue`	`usb.cntl.WValue`
`usbWIndex`	`usb.cntl.WIndex`

To aid with the conversion process, macros with the substitutions are available in the version 1.1 USB.h file. To use the macros, add a define for `OLDUSBNAMES` before including USB.h. It is recommended that you make the actual string changes in the source, because the macro facility is not guaranteed to be available in later versions of the USB.h file.

The `USBClassDriverPlugInDispatchTable` has changed in version 1.1. If the version of `USBClassDriverPluginDispatchTable` is set to

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

`kUSBClassDriverPluginVersion` it indicates that `USBDriverNotifyProcPtr` has the following prototype:

```
OSStatus USBDriverNotifyProc(USBDriverNotification notification, void
                    *pointer, Uint32 refcon);
```

Conventions and Abbreviations

This developer note uses the following typographical conventions and abbreviations.

Conventions

Computer-language text—any text that is literally the same as it appears in computer input or output—appears in `Letter Gothic` font.

Hexadecimal numbers are preceded by a zero x (0x). For example, the hexadecimal equivalent of decimal 16 is written as 0x10.

Note
A note like this contains information that is of interest but is not essential for an understanding of the text. ◆

IMPORTANT
A note like this contains important information that you should read before proceeding. ▲

Abbreviations

When unusual abbreviations appear in this developer note, the corresponding terms are also spelled out. Standard units of measure and other widely used abbreviations are not spelled out.

Here are the standard units of measure used in developer notes:

A	amperes	mA	milliamperes
dB	decibels	μA	microamperes
GB	gigabytes	MB	megabytes
Hz	hertz	MHz	megahertz
in.	inches	mm	millimeters
k	1000	ms	milliseconds

K	1024	μs	microseconds
KB	kilobytes	ns	nanoseconds
kg	kilograms	Ω	ohms
kHz	kilohertz	sec.	seconds
kΩ	kilohms	V	volts
lb.	pounds	W	watts

Other abbreviations that may be used in this note include:

$*n*$	hexadecimal value *n*
ADB	Apple Desktop Bus
ATA	advanced technology attachment
ATAPI	advanced technology attachment packet interface
AV	audiovisual
CD-ROM	compact disc read-only memory
DIN	Deutsche Industries Norm
EMI	electromagnetic interference
GCR	group code recording
IC	integrated circuit
IDE	integrated device electronics
I/O	input/output
IR	infrared
JEDEC	Joint Electronics Devices Engineering Council
PCI	Peripheral Component Interconnect
PIO	parallel input output
SCSI	Small Computer System Interface
SCC	serial communications controller
USB	Universal Serial Bus

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

USB Terminology

The USB terminology used in this document is defined here:

asynchronous data	Data transferred at irregular intervals with no specific latency requirements.
bandwidth	The amount of data capable of being transmitted per unit of time, typically bits per second (bps) or bytes per second (Bps).
big endian	A method of storing data that places the most significant byte of multiple byte values at a lower storage address. For example, a word stored in big endian format places the least significant byte at the higher address and the most significant byte at the lower address. See also, little endian.
bps	Transmission rate expressed in bits per second.
buffer	Storage used to compensate for a difference in data rates or time of occurrence of events, when transmitting data from one device to another. The area in memory where data is either stored or retrieved programmatically.
bulk transfer	Nonperiodic, large bursts of communication typically used for a data transfer that can use any available bandwidth and also be delayed until bandwidth is available.
bus enumeration	Detecting and identifying Universal Serial Bus devices.
class	A group of devices or interfaces that have a set of attributes or functions in common.
client	Software resident on the host that interacts with host software to arrange data transfer between a function in a device and the host. The client is often the data provider and consumer for transferred data.
configuration	One of possibly several settings a device can be programmed into. Configurations may be constrained by available power or bandwidth, or may be differentiated by function. See also, function.

configuring software	The host software responsible for configuring a Universal Serial Bus device. This may be a system configurator or software specific to the device.
control pipe	Same as a message pipe.
control transfer	One of four Universal Serial Bus Transfer Types. Control transfers support configuration/command/status type communications between client and function.
default address	An address defined by the Universal Serial Bus Specification and used by a Universal Serial Bus device when it is first powered or reset. The default address is 0x0.
default pipe	The message pipe created by Universal Serial Bus system software to pass control and status information between the host and a Universal Serial Bus device's Endpoint 0. See also, pipe.
device	A logical or physical entity that performs a function. The actual entity described depends on the context of the reference. At the lowest level, device may refer to a single physical hardware component, as in a memory device. At a higher level, it may refer to a collection of hardware components that perform a particular function, such as a Universal Serial Bus interface device. At an even higher level, device may refer to the function performed by an entity attached to the Universal Serial Bus; for example, a data/FAX modem device. Devices may be physical, electrical, addressable, and logical. When used as a non-specific reference, a Universal Serial Bus device is either a hub or a function.
device address	The address of a device on the Universal Serial Bus. The device address is the default address when the Universal Serial Bus device is first powered or reset. Hubs and functions are assigned a unique device address by Universal Serial Bus software. See also, hub.
device driver	A program responsible for interfacing to a hardware device.

device endpoint	A uniquely identifiable portion of a Universal Serial Bus device that is the source or sink of information in a communication flow between the host and device. See also, isochronous sink endpoint, and isochronous source endpoint.
downstream	The direction of data flow from the host or away from the host. A downstream port is the port on a hub electrically farthest from the host that generates downstream data traffic from the hub. Downstream ports receive upstream data traffic.
endpoint	See device endpoint.
endpoint address	The combination of a Device Address and an Endpoint Number on a Universal Serial Bus device.
endpoint number	A number that identifies a unique pipe endpoint on a Universal Serial Bus device.
frame	The time from the start of one start of frame (SOF) token to the start of the subsequent SOF token. A frame is the master clock of the USB, and is typically 1ms long. See also, SOF.
function	A capability provided to the host by a Universal Serial Bus device. For example, an ISDN connection, a digital microphone, or speakers. A device may provide one or more functions.
host	The computer system in which the Universal Serial Bus host controller is installed. This includes the host hardware platform (CPU, bus, etc.) and the operating system in use.
host controller	The host's Universal Serial Bus interface.
host controller driver	The Universal Serial Bus software layer that abstracts the host controller hardware. Host Controller Driver provides an SPI for interaction with a host controller. Host Controller Driver hides the specifics of the host controller hardware implementation. On the Macintosh this is the Universal Serial Bus interface module (UIM), which is pronounced *whim*.
hub	A Universal Serial Bus device that provides additional attachment points to the Universal Serial Bus.

interface	A collection of pipes which form a logical interface to part or all of a device. USB devices all have an interface or interfaces. Interfaces provide the definitions of the functions available within a device. The device's function or functions are defined by the interfaces it supports. See also, pipe.
isochronous data	A stream of data whose timing is implied by its delivery rate.
isochronous device	An entity with isochronous endpoints, as defined in the USB specification, that sources or sinks sampled analog streams or synchronous data streams.
isochronous sink endpoint	An endpoint that is capable of consuming an isochronous data stream.
isochronous source endpoint	An endpoint that is capable of producing an isochronous data stream.
Isochronous transfer	One of four Universal Serial Bus transfer types. Isochronous transfers are used when working with isochronous data. Isochronous transfers provide periodic, continuous communication between host and device.
little endian	Method of storing data that places the least significant byte of multiple byte values at lower storage addresses. For example, a word stored in little endian format places the least significant byte at the lower address and the most significant byte at the higher address. The USB standard uses little-endian format for multi-byte fields. See also big endian.
message pipe	A pipe that transfers data using a request/data/status paradigm. The data has an imposed structure which allows requests to be reliably identified and communicated. See also, pipe.
packet	Data organized in a group for transmission. Packets typically contain three elements: control information (source, destination, and length), the data to be transferred, and error detection and correction bits.
packet buffer	The logical buffer used by a Universal Serial Bus device for sending or receiving a single packet. This determines the maximum packet size the device can send or receive.

packet ID (PID)	A field in a Universal Serial Bus packet that indicates the type of packet, and by inference the format of the packet and the type of error detection applied to the packet.
physical device	A device that has a physical implementation; for example, speakers, microphones, and CD players.
pipe	A logical abstraction representing the association between an endpoint on a device and software on the host. A pipe has several attributes; for example, a pipe may transfer data as streams (stream pipe) or messages (message pipe).
port	Point of access to or from a system or circuit. For Universal Serial Bus, the point where a Universal Serial Bus device is attached.
root hub	A Universal Serial Bus hub attached directly to the host controller. The root hub is the origin (tier 0) of the USB, and is a software simulation of a standard USB hub device.
root port	The upstream port on a hub.
SOF	An acronym for Start of Frame. The SOF is the first transaction token in each frame. SOF allows endpoints to identify the start of frame and synchronize internal endpoint clocks to the host.
stream pipe	A pipe that transfers data as a stream of samples with no defined Universal Serial Bus structure.
synchronization type	A classification that characterizes an isochronous endpoint's capability to connect to other isochronous endpoints.
transaction	The delivery of service to an endpoint; a complete logical transfer with a beginning and end, consists of a token packet, optional data packet, and optional handshake packet. Specific packets are allowed/required based on the transaction type.
transfer	One or more bus transactions to move information between a software client and its function.
transfer type	Determines the characteristics of the data flow between a software client and its function. Four transfer types are defined: control, interrupt, bulk, and isochronous.

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

UIM	The Universal Serial Bus Interface Module (UIM); the low-level (controller specific) software that provides the upper layers of the USB management software with a hardware abstraction layer to the USB host controller interface hardware.
Universal Serial Bus (USB)	A collection of Universal Serial Bus devices and the software and hardware that allow them to connect the capabilities provided by functions to the host.
USB software	The host-based software responsible for managing the interactions between the host and the attached Universal Serial Bus devices. The USB drivers, USB Manager, and UIM provide these software services on the Macintosh computer.
USB driver	The host-resident software entity responsible for providing common services to clients that are manipulating one or more functions on one or more host controllers, hubs or devices.
upstream	The direction of data flow towards the host. An upstream port is the port on a device electrically closest to the host that generates upstream data traffic from the hub. Upstream ports receive downstream data traffic.

Preliminary Working Draft. Revision 21. © Apple Computer, Inc. 10/26/98

Index

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

Preliminary Working Draft, Revision 21. © Apple Computer, Inc. 10/26/98

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Steve Schwander

ILLUSTRATOR
Dave Arrigoni

Thanks to David Ferguson, Rich Kubota, Barry Twycross, Esmond Lewis, Craig Keithley, Tom Clark, Guillermo Gallegos, Jai Chulani, and Mike Shebanek