



Driver Developer Kit

Mac OS USB DDK API Reference



Preliminary Working Draft, Revision 26

12/23/99

Technical Publications

© Apple Computer, Inc. 1999

Apple Computer, Inc.
© 1999 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

Figures and Tables	11
Preface	About This Note 13
<hr/>	
	Contents of This Note 14
	Supplemental Reference Documents 15
	Mac OS USB Resources 15
	Apple Developer Connection Web Site 16
Chapter 1	Overview 17
<hr/>	
	Introduction to USB 18
	Why Incorporate USB Into the Macintosh Architecture? 18
	Better Device Expansion Model 18
	Compact Connectors and Cables 19
	Use of Standard Hardware 19
	Lower Cost Than Comparable Non-USB Peripherals 20
	Wide Selection of USB Devices 20
	Device Classes 20
	Low- and High-Speed Devices 21
	USB Hub Devices 22
	The USB Root Hub 22
	Compatibility Issues 23
	USB Software Gestalt Selectors 23
	ADB, Serial/LocalTalk, and USB 23
	Macintosh-To-Macintosh USB Connections 23
	USB Storage Devices 24
	Keyboard Requirements 24
	USB Data Transfer Types Supported 24
	USB Controller Support 25
	Maintaining Printer Device and Driver Compatibility 25
	Device Support For Multiple Vendor Specific Devices 26
	USB PCI Adapter Card Support 26

Transaction Timeouts 26

Chapter 2 USB Topology and Communication 29

USB Bus Topology	30
Host Software	30
Physical Topology	31
Logical Topology	31
Communication Over the USB	32
USB Interface	33
USB Devices	33
Endpoints	33
Endpoint 0	34
Non-0 Endpoints	34
Pipes	34
A Look At USB Devices with USB Prober	35
USB Prober Features for Developers	37
USB Prober Windows	37

Chapter 3 USB Software Components 41

Mac OS Software for USB Devices	42
USB Software Presence and Version Attributes	45
USB Interface Module (UIM)	45
USB Manager	46
Hub Driver	47
USB Class Drivers	47
USB Services Library (USL)	48
Applications and USB Drivers	49

Chapter 4 Writing Mac OS USB Drivers 51

Mac OS USB Driver Overview	53
USB Device and Driver Matching	54
Matching Interfaces to Interface Drivers	57
Matching Class Drivers to Composite Devices	59

Device Driver and Interface Driver Matching Differences	60
Core Mac OS USB Driver Data Exports	60
USBDriverDescription Structure	60
USBClassDriverPlugInDispatchTable Structure	65
ValidateHWProc Function	67
InitializeDeviceProc Function	67
InitializeInterfaceProc Function	69
Driver notificationProc Function	70
FinalizeProc Function	71
Handling Hot Unplugging, Dealing With Notifications	71
Communicating With Client Processes	72
The Disappearing Driver	73
Common Ground and The Compatibility Shim	73
Where To Implement a Compatibility Shim	73
Designing A Compatibility Shim	74
Helpful Resources For Compatibility Shim Development	75
Communicating with the Human Interface Device Driver	75
The HIDDeviceDispatchTable Structure	76
The USBHIDModuleDispatchTable Structure	81
Detecting USB Device Presence	83
Mac OS USB Compatibility With Mac OS Toolbox Calls	86

Chapter 5 USB Services Library Reference 89

USB Services Library (USL)	90
Errors And Error Reporting Conventions	91
Device Access Errors	91
Errors on the USB Bus	92
Incorrect Command Errors	93
Driver Logic Errors	93
PCI Bus Busy Errors	93
USB References	93
The USBPB Parameter Block	94
Required USB Parameter Block Fields	98
Standard Parameter Block Errors	99
Using the USBPB For Isochronous Transactions	99
Asynchronous Call Support	101

Polling Versus Asynchronous Completion (Important)	103
Transaction and Data Timeouts	104
USL Functions	106
Determining The Version of USB Software Present	106
USB Configuration Functions	107
Opening An Interface	112
Configuring The Device Interface(s)	113
Finding A Pipe	115
Getting Information About an Open Interface or Pipe	116
Generalized USB Device Request Function	120
USB Transaction Functions	124
Pipe State Control Functions	133
Data Toggle Synchronization	134
Device Control and Status Functions	141
USB Management Services Functions	144
USB Time Utility Functions	146
USB Memory Functions	149
Byte Ordering (Endianism) Functions	152
USL Logging Services Functions	154
USB Descriptor Functions	157
Debugger Aware Flag	163
Deprecated Pipe Functions	163
Constants and Data Structures	164
USB Constants	164
Parameter Block Constants	164
Flag Constants	164
Endpoint Type Constants	165
usbBMRequest Direction Constants	165
usbBMRequestType Type Constants	165
usbBMRequest Recipient Constants	165
usbBRequest Constants	165
Interface Constants	166
Interface Protocol Constants	166
Driver Class Constants	166
Descriptor Type Constants	167
Feature Selector Constants	167
Pipe State Constants	167
USB Power and Bus Attribute Constants	167

Driver File and Resource Types	168
Driver Loading Option Constants	168
Error Status Level Constant	168
USB Data Structures	168
Device Descriptor Structure	169
Configuration Descriptor Structure	169
Interface Descriptor Structure	170
Endpoint Descriptor Structure	170
HID Descriptor Structure	170
HID Report Descriptor Structure	171
USL Error Codes	171

Chapter 6 USB Manager Reference 175

Overview	176
USB Manager API	177
Topology Database Access Functions	177
Getting Device Descriptors	178
Getting Interface Descriptors	178
Finding The Driver For A Device By Class	179
Getting The Connection ID For Class Driver	181
Getting The Bus Reference For a Device	181
Passing Messages To Another Driver	182
Receiving A Message From A Child Driver	183
Registering Shims After Boot Time	183
Adding a Driver For a Device After Boot Time	184
Callback Routine for Device Notification	185
Device Notification Callback Routine	185
Device Notification Parameter Block	186
Installing The Device Callback Request	187
Removing The Device Callback Request	188
Errors Returned By The USB Manager	188

Chapter 7 HID Library Reference 189

Overview	190
----------	-----

HID Library API Reference	192
HID Descriptor Management Functions	192
HID Capabilities Functions	193
HID Report Decoding Functions	200
HID Report Data Encoding Functions	207
HID Library Constants	212
HID Report Constants	213
HIDOpenReportDescriptor Flags	213
HIDGetDeviceInfo Constants	213
HIDOpenDevice Constants	213
HIDInstallReportHandler Constant	214
HIDControlDevice Constant	214
Usage Table Constants	214
HID Library Data Structures	214
HIDDeviceDispatchTable	219
HIS Library Error Codes	220

Appendix A Changes In Mac OS USB Software 223

Major Feature Updates In Version 1.1	223
Improved Bus Enumeration	224
Multiple USB Bus Support	224
Driver Notification Messages	224
Isochronous Transfer Support	225
Improved Functionality For USB Control Requests	225
Code Changes Required To Support The Version 1.1 USBPB	226
Major Features Introduced In Version 1.2	227
Changes and Enhancements Introduced in Version 1.4	228
Release Notes And Compatibility Issues	230
Bulk Data Transfer Performance Issues	230
Understanding Generic Drivers	231
Core 99 CPU Issues	232
Audio Support in Version 1.3.5 and 1.4	232
USB Extensions For Development and ROM Driver Conflicts	233

Appendix B Conventions and Abbreviations 235

Conventions	235
Abbreviations	235

Appendix C USB Terminology 237

Index	243
-------	-----

Figures and Tables

Chapter 1	Overview	17	
	Table 1-1	Examples of USB device classes	20
Chapter 2	USB Topology and Communication	29	
	Figure 2-1	USB physical topology	31
	Figure 2-2	USB communication flow	32
	Figure 2-3	USB Prober utility, USB Bus Devices window	35
	Figure 2-4	USB Prober view of a USB device	36
	Figure 2-5	Expert log window	38
	Figure 2-6	USB Prober Device Recorder window	39
Chapter 3	USB Software Components	41	
	Figure 3-1	USB architecture	44
Chapter 4	Writing Mac OS USB Drivers	51	
	Figure 4-1	Device driver matching algorithm	55
	Figure 4-2	Interface driver matching algorithm	58
Chapter 5	USB Services Library Reference	89	
	Table 5-1	Standard parameter block errors	99
	Table 5-2	Error definitions	171
Chapter 6	USB Manager Reference	175	
	Figure 6-1	Device addition event sequence on the USB	176
	Table 6-1	USB Manager error codes	188

Chapter 7 HID Library Reference 189

Figure 7-1	HID library in USB software architecture	191
Table 7-1	HID library error codes	220

About This Note

This document provides an 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 related to native drivers 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 237 before moving on to the rest of the material in this document.

If you are interested in finding out about the features USB provides and want to get a basic description 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,” Chapter 4, “Writing Mac OS USB Drivers,” the reference material in Chapter 5, “USB Services Library Reference,” and Chapter 6, “USB Manager Reference.” In addition, look at the example code provided in the Mac OS USB Device Driver Kit.

Major differences between versions of the Mac OS USB software are noted in Appendix A, “Changes In Mac OS USB Software,” page 223. 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. Versions 1.2, 1.3, and 1.4 of the Mac OS USB software are also covered by the APIs defined in this document.

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, “Writing Mac OS USB Drivers,” discusses the composition of Mac OS USB drivers and provides details about how the Mac OS USB software communicates with the driver. It also discusses USB compatibility shims, which developers can create to provide a layer of software that allows applications or other client processes to communicate directly with their vendor specific USB drivers through exported APIs.
- Chapter 5, “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 6, “USB Manager Reference,” describes the Mac OS USB Manager library that provides service to the Mac OS and extension clients.
- Chapter 7, “HID Library Reference,” defines the HID library API.
- Appendix A, “Changes In Mac OS USB Software,” provides information about the differences between various releases of the Mac OS USB software that developers need to be aware of.
- 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, Mac OS Power PC native drivers, the Mac OS Name Registry, the Drivers Services Library, and other invaluable services for the development of modern device drivers for the Mac OS platform, refer to *Designing PCI Cards and Drivers for Power Macintosh Computers*.

To understand the Mac OS APIs and services provided for application programmers, you 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/>

The information found in *Inside Macintosh: Power PC System Software, Chapter 3, "Code Fragment Manager"* is a handy reference for developers writing USB device and interface drivers, because USB drivers are essentially code fragments. The information there is not specific to USB device drivers, but it does define how code fragments work, and provide descriptions of the APIs that support code fragments in the Mac OS environment.

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 the latest technical and marketing information specifically for developing successful Macintosh-compatible software and hardware products. Developer World can be reached at

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

Overview

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

Overview

addition of external USB multiport hubs. Hubs can also be 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 1-22) 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 a 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

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. A hub may also be embedded in another device, such as a keyboard or display.
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, 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 2-31).

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. Additional information about feature compatibility between each of the Mac OS USB system software releases can be found in Appendix A.

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 Open Firmware 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 Open Firmware. 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.

Overview

- 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.

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.

Overview

- 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.

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.

Maintaining Printer Device and Driver Compatibility

This section deals with issues related to USB printer dongles, USB printers, and USB printer device driver development. What follows is a list of recommendations that ensure compatibility and provide a quality customer experience when using a USB printer on a Macintosh computer.

- Put your vendor ID code in a EPROM of your USB printer dongle adaptor or printer. Don't use the default ID that comes with the USB device part. Failure to do so may cause your vendor specific driver to match against a device other than your own. For example, without a vendor ID EPROM, the USB software will see your dongle as a USB to parallel bridge device rather than your printer device, and your printer driver may never be matched with the printer.
- If you do supply your own printer driver, it absolutely must be vendor specific and not generic. It must be named differently from the Apple USB printer driver, and match to your vendor ID.

There are also important issues related to driver loading options in the driver description structure that must be followed. The driver description structure is defined in "USBDriverDescription Structure" (page 4-60). Essentially, the rule is that you must set your driver loading options so that your driver does not match to generic devices. To set your driver loading options to do not match generic, specify `kUSBDoNotMatchGenericDevice` in the `USBDriverLoadingOptions` field of the `USBDriverDescription` driver description structure.

Overview

Additional information about USB drivers and driver loading can be found in Chapter 5, “USB Services Library Reference,” and “Installing The Device Callback Request” (page 6-187).

Device Support For Multiple Vendor Specific Devices

Beginning with version 1.2 of the Mac OS USB software, multiple class drivers can be merged into a single vendor specific USB extension file. Details on how this is done and the benefits that are provided by this feature can be found in “Major Features Introduced In Version 1.2” (page A-227).

As of version 1.4 of the Mac OS USB software both shims and class drivers can be merged into a single USB extension file. Prior to version 1.4, it was only possible to merge class drivers in a single file.

As stated above it version 1.2 provides support for multiple drivers in a driver file. Unfortunately, this does not include support for merging a shim into the same file as a class driver. Shim code is found because its creator code is 'usbs'. A driver file is found because its creator is 'usbd.' If you attempt to merge both shims and class drivers into a single extension running on versions 1.2 through 1.3.x of the Mac OS USB software, either the shim or the driver will not be recognized, depending on the creator type of the resulting file.

For USB 1.4, it will be possible to merge the shim and the driver into a single file and to set the creator code as either 'usbs' or 'usbd'.

USB PCI Adapter Card Support

Mac OS USB software is available to support USB PCI adapter cards. The USB system extensions in this package allow users of Power Macintosh computers with PCI slots to install a third-party USB PCI adapter card and communicate with USB devices connected to the USB ports on the adapter card. The software can be found at the Apple Software Updates:

<http://asu.info.apple.com/swupdates.nsf/artnum/n11487>

This software does not install on Macintosh computers with built-in USB ports.

Transaction Timeouts

Beginning with version 1.3 of the Mac OS USB software, a 5-second timeout is enforced on control transactions. This can be overridden for drivers with

CHAPTER 1

Overview

devices that do not support the new behavior. A facility has also been added for class drivers to set a no data timeout for devices. See “Transaction and Data Timeouts” (page 5-104) for details about these new features in the Mac OS USB software.

CHAPTER 1

Overview

USB Topology and Communication

This chapter introduces the USB topology and how the USB host software communicates with devices. 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 5, “USB Services Library Reference,” for additional information about the contents of the device configuration descriptor structure.

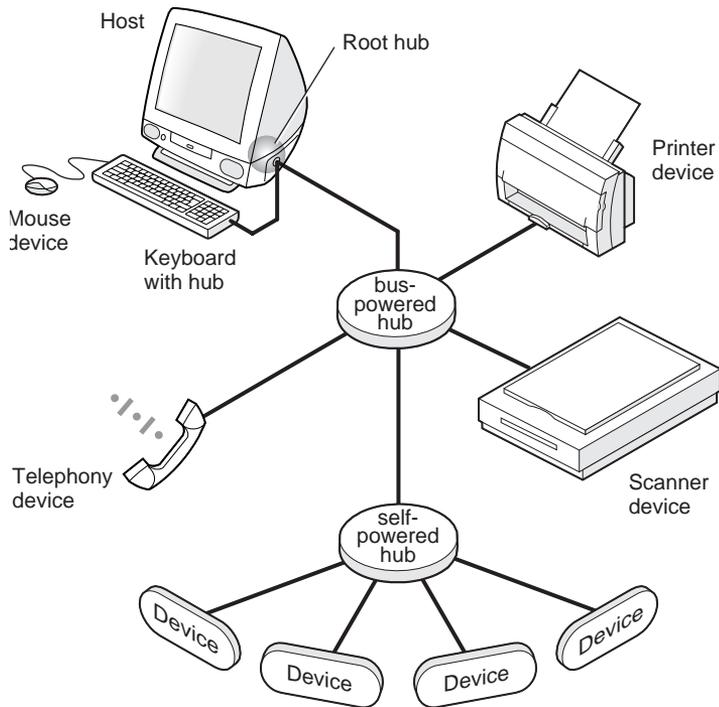
The host interacts with USB devices through the host controller. The host is responsible for:

- 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

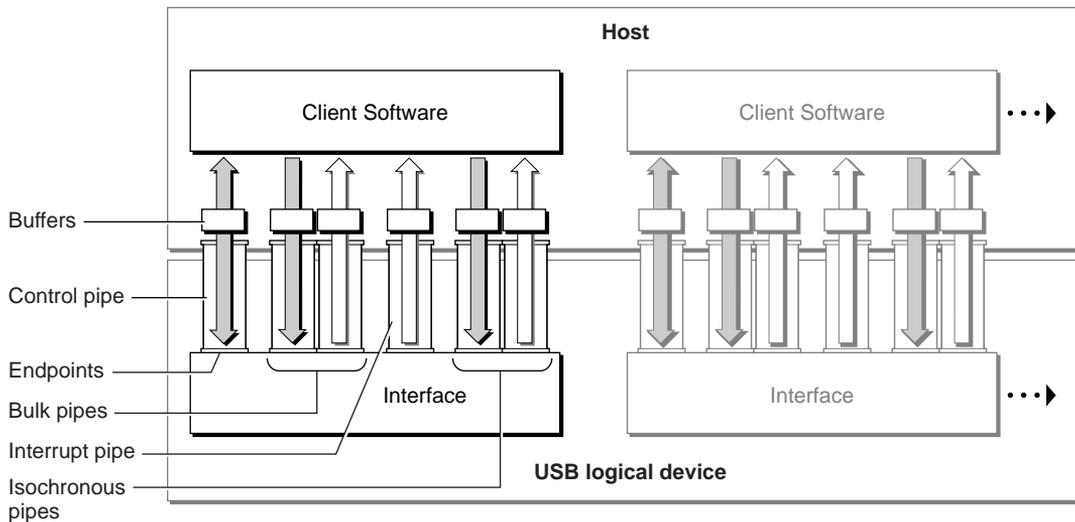
seen as a linear addressing space. The host is aware of the physical topology so that it can accurately support connection and disconnection on hubs with attached devices.

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 it 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. The interface descriptor is defined on (page 5-170).

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 device 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.

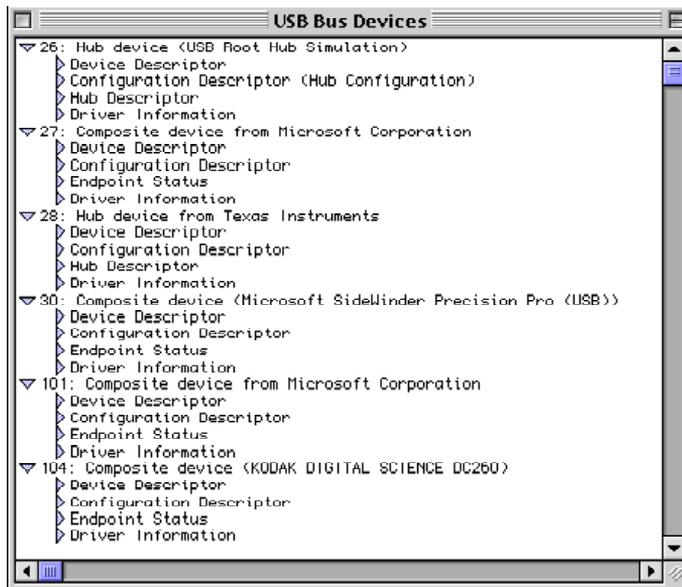
For a detailed descriptions of USB interfaces, endpoints, pipes, communication modes, and transfer types see the Universal Serial Bus Specification, which can be found at <<http://www.usb.org>>.

A Look At USB Devices with USB Prober

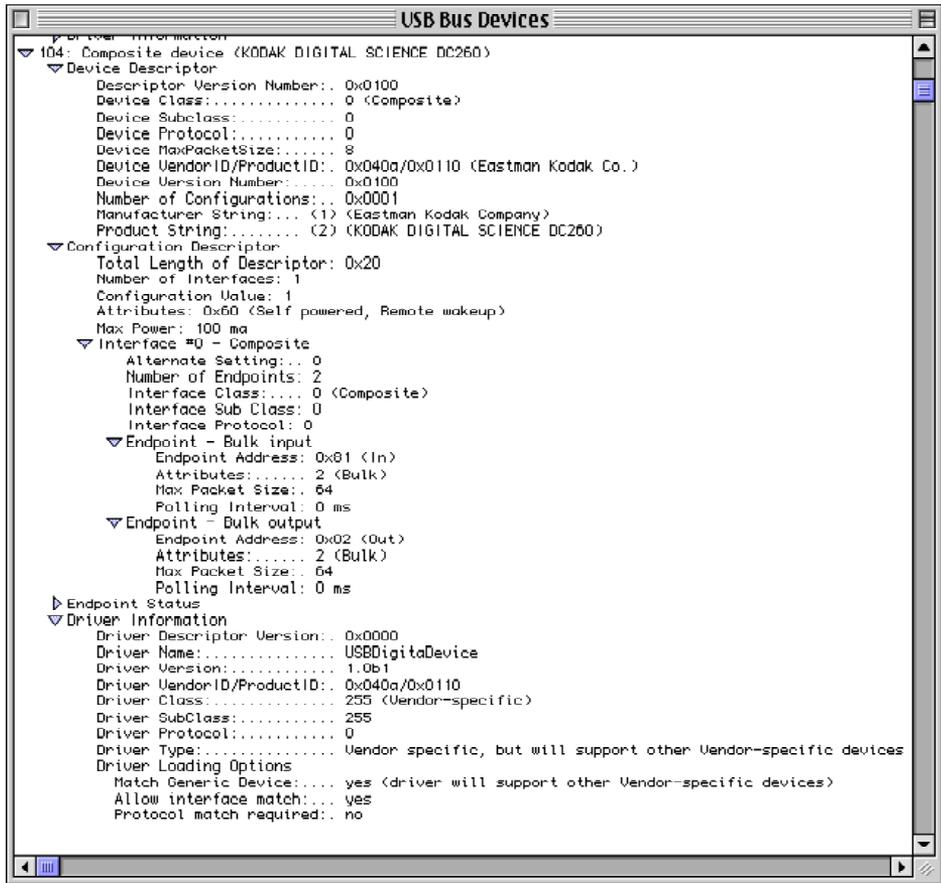
The USB Prober application, which is included in the Utilities folder of the Mac OS USB Device Driver Kit, is a utility for examining devices on the USB.

Figure 2-4 shows the USB Bus Probe window. The example shows the root hub simulation at the top, then various USB devices connected to a 2-port USB PCI card. A mouse and a self-powered 4-port hub are connected to the USB PCI card. The 4-port hub has three devices connected to it.

Figure 2-3 USB Prober utility, USB Bus Devices window



In Figure 2-4 the device descriptor, configuration descriptor, and driver information categories for a digital camera with a USB port are expanded to show how the device and its associated drivers are defined.

Figure 2-4 USB Prober view of a USB device

The digital camera belongs to the composite class and has a composite interface with a two bulk transfer endpoints; one supporting bulk input and the other supporting bulk output transfers. The camera's device driver is matched to the vendor ID and product ID values found in the device descriptor and driver descriptor.

USB Prober Features for Developers

The USB Prober application provides several features that are useful to programmers who are debugging USB device drivers, or anyone interested in knowing what devices the Mac OS USB software recognizes on the USB.

The primary features of the USB Prober are those found in the Windows menu.

USB Prober Windows

The features of the USB Prober Windows menu include:

- **USB Bus Probe:** This window option lists all the device and driver information known about USB bus devices connected to the USB, including the root hub. The content typically displayed in the USB Bus Device window is shown in Figure 2-3 and Figure 2-4.
- **Open HCI Registers:** This window option displays the OHCI controller for each USB bus in the current system, and displays a decoded hierarchical list of the contents of the controller registers.
- **USB Expert Log:** This window option, shown in Figure 2-5, provides a running list of status messages posted by various parts of the USB system software. Both the USB Expert and device driver software routinely post status information here, giving a great deal of insight as to what's happening, or not happening on the bus. Information in this log can help you to determine when or if your driver is loading as expected. You can control the level of status that is displayed in the window by selection one of 5 levels found under the Commands menu in the Status Level menu item. The levels are defined as:
 - 1: Fatal errors.
 - 2: General errors that may or may not effect operation.
 - 3: General driver messages.
 - 4: Important messages generated by the USB Expert and USL.
 - 5: General messages from the USB Expert and USL. This is the default level selected when the Prober is first launched.

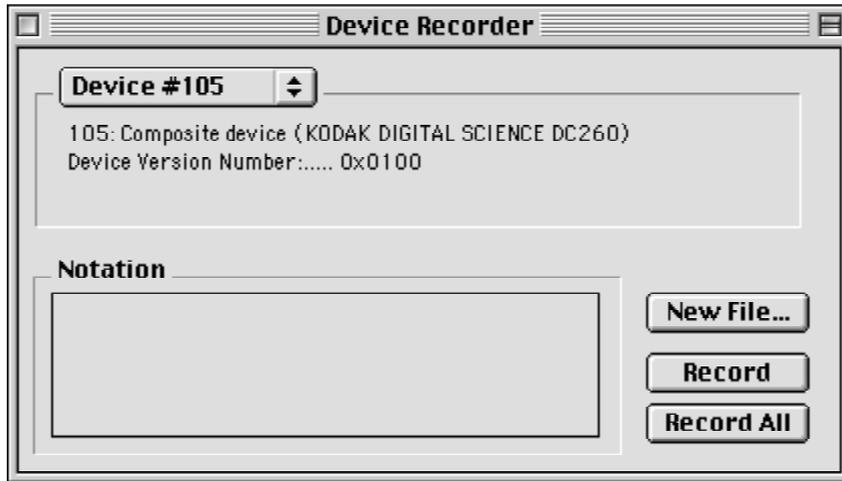
Figure 2-5 Expert log window

```

USB Expert Log
1.294: [Expert] Initializing USB Expert 1.1
1.310: [Driver] InstallDeviceNotification Request
1.310: [Driver] InstallDeviceNotification Request
1.310: [Driver] Status: USL - Adding bus index:
1.330: [Driver] Status: USL - New device: : 108
1.330: [Expert] Closecall 50 ticks
1.330: [Driver] Status: USL - New device power: : 250
1.361: [Expert] Add device class: 9 subclass: 1
1.486: [Expert] ***LoadClassDriver***
1.488: [Expert] Class driver located in ROM: USBHub1Apple
1.788: [Driver] Status: Externally powered Hub attached - power supply good
3.050: [Expert] Bus Enumeration Took 1733 Milliseconds
3.050: [Expert] Early termination was avoided by 60 ticks
11.885: [Expert] *****
11.885: [Expert] No composite devices found
11.885: [Expert] Checking disk for shims...
11.892: [Expert] Checking disk for drivers...
12.897: [Expert] Bus Enumeration Took 1016 Milliseconds
12.897: [Expert] Early termination was avoided by -1 ticks
12.897: [Expert] Finished checking disk.
31.778: [Driver] InstallDeviceNotification Request
31.807: [Manager] ExpertNotificationProc: GetNextDeviceByClass

```

- Device Recorder:** This window option records information about a specified device from a list of attached devices is provided in a popup menu. When you press the Record button in the Device Recorder window, shown in Figure 2-6, USB Prober saves the entire list of information in the USB Bus Devices window that corresponds to the selected device into a file on disk. The information it saves to disk includes the raw device, configuration, string, and hub descriptor data.

Figure 2-6 USB Prober Device Recorder window

Device Record Viewer: Device specific information saved by the Device Recorder can be displayed in hierarchical view with the Device Record Viewer. The information can be saved as a text file that can be viewed by any application that supports text files. The list of information about the device is fully expanded in the text file that is saved.

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

CHAPTER 2

USB Topology and Communication

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 5, “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. Other drivers have been added in later releases.

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. As of the last update to this document, version 1.3.5 of the Mac OS USB software supports third-party USB PCI cards. The USB Device Extension and USB Support extension support software development and third-party USB PCI cards. The HID Library and SerialShimLib shared libraries are also required in the Extensions folder to support the latest versions of Mac OS USB software.

The USB Device Extension contains the following class drivers

- USBCompositeDriver - A class driver which is loaded for class 0 and subclass 0 devices (typically keyboards & mice, although a few printers and modems appear as composite class devices).
- USBHubDriver0 & USBHubDriver1 - The Hub driver for the root hub, the Apple USB keyboard's hub, and most 3rd party hubs.

USB Software Components

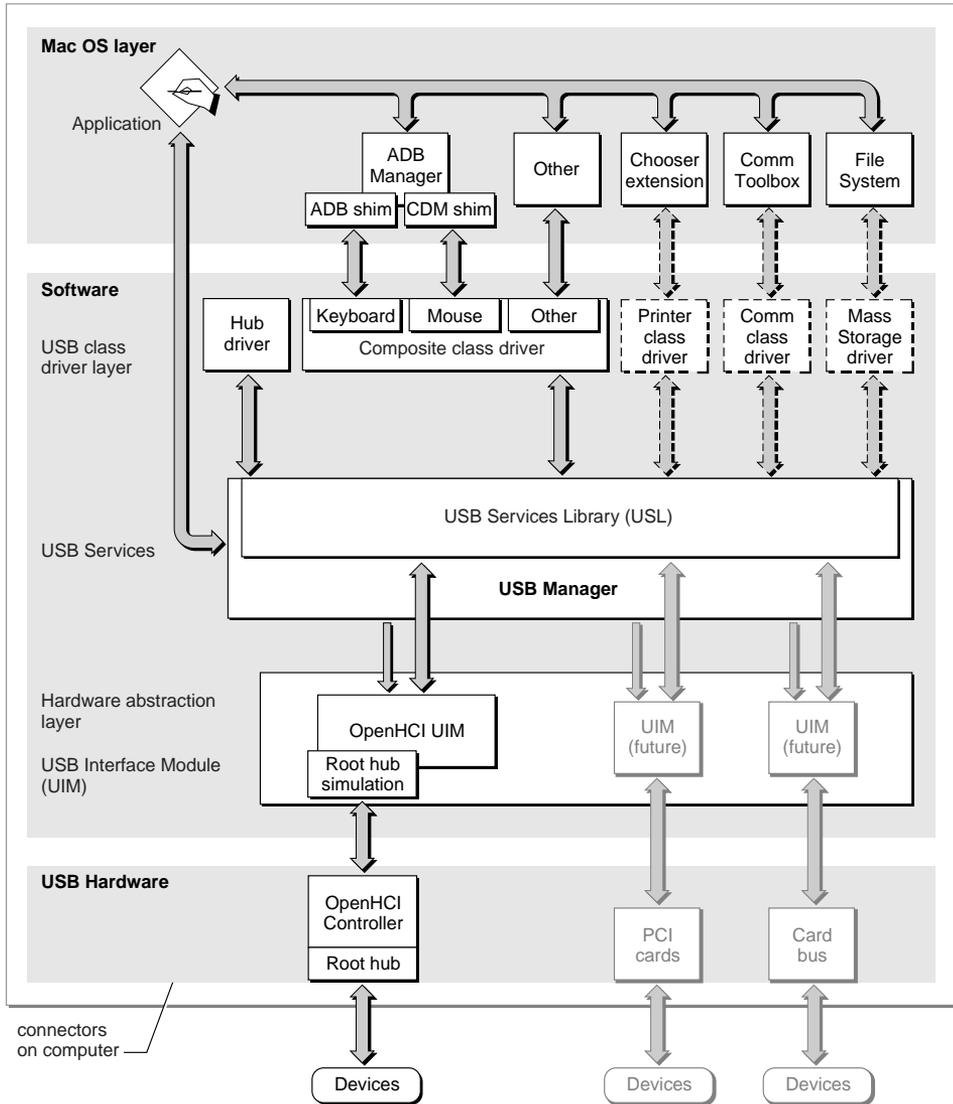
- USBKeyboardModule - The HID software module loaded for a keyboard interface (typically in a compound device)
- USBMouseModule - A HID software module loaded for a mouse interface (typically in a compound device).
- USBHIDUniversalModule - A HID module designed to support a number of 3rd party gaming devices via Game Sprockets. Refer to the “Mac OS USB Compatibility Notes” in the DDK for a description of the game devices supported.
- USBOpenHCIDriver - The HAL (Hardware Abstraction Layer), also known as the UIM (USB Interface Module). This driver provides an abstracted interface to the OpenHCI host controllers on Mac motherboards and PCI controller cards.

USB Support extension is the INIT (actually an 'expt') portion of the Family Expert. Loaded prior to the INIT parade, it starts the “find a USB bus” (UIM driver), loads the hub driver (for the root hub), and starts the hub driver running. Once running, the hub driver locates devices attached to the port and asks the FamilyExpertLib to load drivers for the devices connected to the hub's ports.

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 file, Mac OS 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. Later releases will provide support for other devices.

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 and revision number. The current value for the minor version number is 2. This number will increment with each minor release.
RR	The next byte contains the release stage. The release stage is defined as: 0x20 = development, 0x40 = alpha, 0x60 = beta, and 0x80 = final. If the software was at the beta release stage, this number would be 0x60.
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 for Power Macintosh Computers." 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.

Open Firmware 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 6, "USB Manager Reference." The API for the USB Services Library is described in Chapter 5, "USB Services Library Reference."

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 implemented as Shared Libraries with a file type 'ndrv' and creator 'usbd'. As with other PCI drivers, the code fragment Manager (CFM) code fragment must export driver description structures to characterize the USB functionality the driver provides. The USB Manager uses the description structure to match drivers 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.

Mouse cursor movement and other operations reported to the Mac OS from keyboards and mice are handled by compatibility shims. Shims provide a layer of software translation between the USB driver and upper-level APIs, such as the Cursor Device Manager (CDM) in the case of a mouse or pointing device. An ADB shim provides compatibility services for the ADB Manager to support the USB keyboard. The ADB compatibility shim services allow older applications that are not USB aware to function as expected with a USB keyboard. The CDM shim provides the same level of compatibility with the USB mouse for older applications.

For additional information about how compatibility shims are used in Mac OS USB environment, see “Applications and USB Drivers” (page 3-49), the driver code sample available in the Mac OS USB DDK, and Chapter 4, “Writing Mac OS USB Drivers.”

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 5, “USB Services Library Reference.”

Applications and USB Drivers

The Mac OS USB architecture implements a driver model in which the USB drivers are dynamically loaded libraries, code fragments with exported symbols in the form of a driver description and driver dispatch table. This allows USB driver writers to export whatever public or private APIs they see fit for their devices. The symbols that drivers export are defined in “Core Mac OS USB Driver Data Exports” (page 4-60).

The USB Manager and its sibling, the USB Family Expert, are used to manage the loading and unloading of USB drivers. The USB Family Expert does the matching of drivers to devices, while the USB Manager provides APIs that let applications, compatibility shims, INITs, and so on, discover what USB devices are currently attached to the USB, and request to be notified whenever devices are connected or disconnected.

There are a number of ways that applications can communicate with USB drivers. They range from calling directly into the USB driver, creating a layer between the application and class driver (a compatibility shim), or creating a unit table driver that calls the class driver. You could also have cdevs and INITs call class drivers, as long as you handle the mixed mode environment properly.

The decision to use one method over the other depends on the following factors:

- Does your device driver emulate a legacy I/O device? For example, a serial device.
- Do you need to have something (a shim) manage multiple devices connected simultaneously?
- Are you planning on publishing the APIs to application programmers, or will you be the only application writer?
- Do you care about architectural issues that may affect you in the future?

Ideally application writers should not have to worry about the USB Manager. Applications have a way of living on for 5, 10, or 15 years. And who knows what changes may take place in the USB Manager services over that entire time span. Instead of relying specifically on the USB Manager APIs from the start, it would be better to export a set of abstract functions from your driver that have little to do with USB specifically.

USB Software Components

For example, put your exported APIs into a USB compatibility shim, and let your application call the compatibility shim. That way, you can revise your USB compatibility shim to support any revisions to the USB Manager without needing to rewrite an entire application.

And if you're emulating a legacy device that has a driver model already established (for example, .ain, .aout, .sony, and so on), then you'll want to create a unit table driver that applications can talk to. That's what is done with the Apple USB Storage class driver and USB Modem driver, which are available in Mac OS USB software version 1.2. They both have compatibility shims that create standard Device Manager style unit table drivers.

You need to be very careful when having applications call class drivers directly. It's possible that the driver could unexpectedly quit, or the device may be unexpectedly unplugged. You need to pay careful attention to application quitting and device removal situations.

In summary:

If you're planning on creating APIs that 3rd party application writers can use, then abstract your API so that it isn't USB specific.

If you're planning on developing a device that you will emulate a legacy unit table driver, then you'll need to create unit table driver that in turn calls your USB device's class driver.

You can also have your application call directly into your class driver. But you'll need to weigh the long term application compatibility issues and see what makes the greatest amount of sense to you.

Writing Mac OS USB Drivers

This chapter describes how the USB Manager interacts with class drivers when a USB device is recognized, and defines the core functionality that the driver must implement to operate properly on the Macintosh platform.

This chapter provides information about the composition of a Mac OS USB driver, describes how drivers are matched to devices, defines the driver entry points and core data structures drivers must export, and discusses how a driver gets called by the Mac OS USB software. Sample code is also provided where necessary to support important concepts.

This chapter is divided into the following sections:

- “Mac OS USB Driver Overview” deals with the basic composition of a USB driver, and refers to the symbols, defined later in “Core Mac OS USB Driver Data Exports”, that a driver must export to describe its device support features to the Mac OS USB software.
- “USB Device and Driver Matching” describes the criteria used by the Mac OS USB software to match drivers to USB devices. This section also describes how the fields in the driver core data exports are used to facilitate accurate matching of drivers to devices.
- “Core Mac OS USB Driver Data Exports” defines the contents of the driver description structure that the Mac OS USB software uses to match and load USB class drivers, and the driver dispatch table that class drivers use to communicate with the Mac OS USB software and monitor device activity on the USB. The structures defined in this section are the two symbols that a driver must export to the Mac OS USB software to operate properly with USB devices on the Macintosh platform.
- “Handling Hot Unplugging, Dealing With Notifications” further defines how notification messages, listed in “Driver notificationProc Function” (page 4-70), are used by Mac OS software to communicate device activity on the USB to USB drivers.
- “Communicating With Client Processes” describes how to facilitate communication between Mac OS applications and processes and USB class drivers. Design and use of compatibility shims is discussed in this section.
- “Detecting USB Device Presence” provides ideas about how to implement two different means for finding a USB device.
- “Mac OS USB Compatibility With Mac OS Toolbox Calls” discusses issues USB class drivers should be aware of with regard to the services provided by the Mac OS Toolbox.

Apple provides class drivers for USB hub devices, HID devices (mice, keyboards, joysticks, and gamepads), printing, communications, and mass storage devices shipped in specific system configurations. Other drivers are provided as samples in the Mac OS USB DDK. Unless devices are 100 percent compliant with the class specification and are supported by Apple, developers need to write a class driver for their device.

In addition to writing a USB class driver, another important software component to consider creating is a software compatibility shim which makes it possible for existing processes, such as a Comm Tool box module (CTM) or Chooser printer device, to communicate with the USB device.

Note

Input Sprockets version 1.7 provides support for HID devices, such as joysticks, gamepads, steering wheels, and rudder pedals. Input Sprockets version 1.7 will be incorporated into future releases of the Mac OS software.

Mac OS USB Driver Overview

USB class drivers provide the software interface between USB hardware and client processes that communicate with the hardware. As discussed in Chapter 3, “USB Software Components,” a Mac OS USB class driver is implemented as a shared library with a file type 'ndrv' and creator 'usbd'. In a similar fashion to PCI drivers, the driver CFM code fragment must export driver description structures to characterize the USB functionality the driver provides. All USB class drivers are recognized by the contents of the `USBDriverDescription` structure, which is defined in, “USBDriverDescription Structure” (page 4-60). The use of the `USBDriverDescription` structure for Mac OS USB drivers is modeled after the `DriverDescription` structure discussed in “Designing PCI Cards Drivers for Power Macintosh Computers, Revised Edition.”

The Mac OS USB Manager handles the recognition of all connected USB devices, both at startup and when hot-plugged. As part of device recognition, Mac OS USB Manager assigns the unique USB address, and opens a control pipe to endpoint zero of the device. Mac OS USB Manager queries the device's endpoint 0 for the device descriptor, which includes information such as vendor id, product id, device class, subclass, and protocol.

Upon receipt of the standard device descriptor information, the Mac OS USB Manager searches for all USB class drivers by looking at files of type 'ndrv' and creator 'usbd'. For each file, the USB Manager searches for all code fragments and for the exported symbol `USBDriverDescription` and subsequently at the first four bytes of the structure to match the `kUSBDriverDescriptionSignature`. This constant identifies the CFM code fragment as a USB class driver. Refer to the section “USBDriverDescription Sample” for more information and an example of the content of the `USBDriverDescription` structure.

Note

In version 1.0 through 1.1 of the Mac OS USB software only a single USB code fragment can exist in the driver file. Beginning with version 1.2 multiple driver CFMs within a driver file are supported. For more information about how to support multiple USB devices with similar characteristics, see “Major Features Introduced In Version 1.2” (page A-227).

USB Device and Driver Matching

Mac OS USB defines a match criteria to determine which class driver is selected to support a USB device. A match value is assigned to each USB driver. The following criteria determine the match value under Mac OS USB. The match criteria are listed from highest to lowest.

1. The vendor and product IDs match. A match of the `usbDeviceReleaseNumber` increases the match value.
2. The device class vendor specific, match of vendor ID, device subclass and protocol.
3. `kUSBDoNotMatchGenericDevice` bit clear in the `usbDriverLoadingOptions` field of driver description structure, match of device class, device subclass and device protocol.
4. `kUSBDoNotMatchGenericDevice` bit clear in the `usbDriverLoadingOptions` field of driver description structure, match of device class, device subclass and `kUSBProtocolMustMatch` bit clear in the `usbDriverLoadingOptions` field of driver.

Figure 4-1 provides a flow diagram that illustrates how the USB Manager assigns values to the driver matching criteria.

Figure 4-1 Device driver matching algorithm



Typically a driver should set the `kUSBDoNotMatchGenericDevice` bit in the `usbDriverLoadingOptions` field of the `USBDriverDescription`, defined in

“USBDriverDescription Structure” (page 4-60), to indicate that it only supports vendor specific devices. This way the driver is not matched as a generic driver for a device that it may not actually support. If the bit is clear, then the driver indicates generic support for devices by device class and subclass. Unless you plan to write a class driver that supports all devices with a specific class and subclass, you should have your device return a device class of 0xFF (vendor specific) and set the `kUSBDoNotMatchGenericDevice` bit in the `usbDriverLoadingOptions` field of the `USBDriverDescription` structure.

Regardless of whether a match, or “best match” is found, the USB Manager registers the device in the Name Registry. Drivers that could potentially support the device are accessed in a high-to-low rank order through the driver’s `USBClassDriverPluginDispatchTable`.

The driver’s `USBClassDriverPluginDispatchTable` defines five functions that a Mac OS USB driver must support. The contents of the structure and functions are defined in the section “USBClassDriverPlugInDispatchTable Structure” (page 4-65). The functions in the `USBClassDriverPluginDispatchTable` are the driver entry points that the Mac OS USB uses to properly instantiate the appropriate driver for a USB device.

The USB Manager calls the `validateHWProc` to have the class driver verify that the driver supports the hardware. The `validateHWProc` routine can compare the vendor and device ID, or other fields of the device descriptor, to validate that it can support the device. If no error is returned, the `initializeDeviceProc` routine is called so that the class driver can establish communications with the device. Refer to the sections “ValidateHWProc Function” and “InitializeDeviceProc Function” below for more information on implementing these functions.

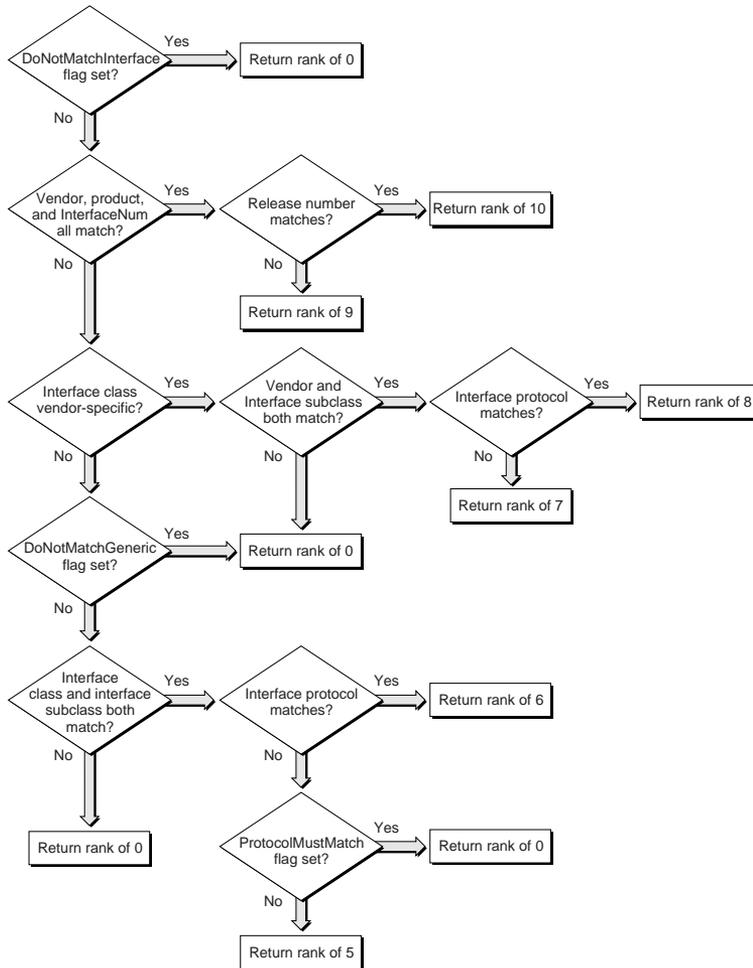
For USB devices that match to the composite class type (device class 0, subclass 0), the Apple USB composite class driver is loaded, which in turn selects from the available configurations. The composite class driver sets the appropriate configuration based on the power and or bandwidth available, and examines the interface descriptors by calling the `USBExpertInstallInterfaceDriver` function for each interface. The USB Manager uses the interface descriptor, class, subclass, and protocol, and rescans the USB class drivers for a match of the interface descriptor section of the driver descriptor. When a match (or best match) is found, the USB Manager loads the driver code fragment, and finds the `USBClassDriverPluginDispatchTable` for the matched driver. The `initializeInterfaceProc` function is called for the interface driver to initiate communication with the particular function in the device.

Matching Interfaces to Interface Drivers

Similar to matching devices with drivers, Mac OS USB defines a match criteria to determine which class driver is selected to support a USB interface. A match value is assigned to each USB driver. The following criteria determine the match value under Mac OS USB v1.0. The criteria is listed in order of match value, 10 being the highest or best match.

1. vendor and product IDs match along with the configuration value and interface number. A match of the `usbDeviceReleaseNumber` increases the match value by 1.
2. interface class vendor specific, match of vendor ID and interface subclass. A match of the protocol increases the match value by 1.
3. `kUSBDoNotMatchGenericDevice` bit clear in the `usbDriverLoadingOptions` field of driver, match of interface class, interface subclass and interface protocol.
4. `kUSBDoNotMatchGenericDevice` bit clear in the `usbDriverLoadingOptions` field of driver, match of interface class, interface subclass, and `kUSBProtocolMustMatch` bit clear in the `usbDriverLoadingOptions` field of driver.
5. `kUSBDoNotMatchInterface` bit set in the `usbDriverLoadingOptions` field of driver - don't match driver for interface.

Figure 4-2 provides a flow diagram that illustrates how the USB Manager assigns values to the driver matching criteria.

Figure 4-2 Interface driver matching algorithm

Once the USB Manager has matched a class driver to the interface, the driver's `initializeInterfaceProc` function pointed to from the driver dispatch table is called. The `initializeInterfaceProc` function is used by the class driver to prepare itself to communicate with the interface. When interface class drivers are loaded, the `validateHWProc` is not called. However, a driver can return an error from the `initializeInterfaceProc` and the next best matching driver is

loaded. As with device matching, you want to specify matching to interfaces by the vendor, product ID, and interface number, or by setting the interface class to `0xFF` (vendor specific), and setting the `kUSBDoNotMatchGenericDevice` bit in the `usbDriverLoadingOptions` field to indicate that the driver does not support a class and subclass interface generically.

When a device has been disconnected from the bus, the driver's `notificationProc` function is called, and when it returns `noErr`, the driver's `finalizeProc` function is called by the USB Manager. The `finalizeProc` function is called at task time so that a class driver can clean up memory allocations, and perform other operations associated with the driver or interface no longer being present. The `finalizeProc` must be synchronous. That is, when the driver returns `kUSBNoErr`, the Mac OS USB software assumes that there is no further need to keep the driver code fragment around, and unloads it from memory.

Matching Class Drivers to Composite Devices

For composite class USB devices, the Mac OS USB software implements class driver matching differently in the following two cases

- when the device is connected at startup, and
- when the device is “hot plugged”

The following explanation of composite class device and driver matching applies to USB v1.0.1 and earlier. At system startup, the Mac OS USB software cannot access file based drivers. To support boot devices, such as the keyboard and mouse, which are USB composite HID class devices, the Mac OS USB software uses the generic composite class driver in the Mac OS ROM file (in iMac and later computers). If a device is identified as a composite device, and it is attached to the USB bus at startup, then the generic composite class driver is matched to that device. This happens even though there may be a file based driver which is a better match for your device. Once the driver is matched, USB v1.0.1 does not check for a better match when the file system becomes available.

If the composite device is hot plugged later, then the USB software can find a better matching disk-based driver, if one exists.

Mac OS USB software v1.1 and later look for a better match disk-based driver when the file system becomes available.

The generic composite class driver sets the configuration for the composite class device to the first configuration it finds. The generic composite class driver only

checks that the configuration's power requirement is within the available power limits of the hub to which the device is connected.

Device Driver and Interface Driver Matching Differences

The implementation of a Mac OS USB class driver is such that drivers may be loaded as either device drivers or interface drivers. The primary difference between device and interface driver loading is that the device driver code must issue a `SetConfiguration` call (a `USBDeviceRequest` call with the `usbRequest` field set to `kUSBRqSetConfig`) to define which configuration is active. In contrast, the interface driver can not set a configuration. Both device and interface drivers identify the interfaces present, find one to activate, open the pipes in the desired interface, and handle the I/O on the pipes within the interface.

Typically, the class driver is loaded as an interface driver if the device is detected as a composite class device and the Apple USB composite driver is loaded. The Apple USB composite class driver detects the available configurations and makes the `USBExpertInstallInterfaceDriver` call to the USB Manager, which results in the appropriate class driver being loaded and initialized as specified in the driver's `initializeInterfaceProc`, defined in "USBClassDriverPlugInDispatchTable Structure" (page 4-65).

Core Mac OS USB Driver Data Exports

This section defines the contents of the `USBDriverDescription` and the `USBClassDriverDispatchTable` structures. The two structures described here are the symbols that a Mac OS USB driver must export to communicate with the Mac OS USB software and in turn the USB device and/or interface the class driver module supports.

USBDriverDescription Structure

The `USBDriverDescription` is a required exported symbol that the USB Manager uses to first identify USB class drivers, and second, to identify whether a driver supports a specific USB device. The structure of the `USBDriverDescription` structure is as follows:

Writing Mac OS USB Drivers

```

struct USBDriverDescription
{
    OSType                usbDriverDescSignature;    /* Signature field of */
                                                         /* this structure. */
    USBDriverDescVersion  usbDriverDescVersion;     /* Version of this */
                                                         /* data structure */
    USBDeviceInfo         usbDeviceInfo;            /* Product & Vendor Info */
    USBInterfaceInfo     usbInterfaceInfo;         /* Interface info */
    USBDriverType         usbDriverType;           /* Driver Info */
    USBDriverLoadingOptions usbDriverLoadingOptions; /* Options for class */
                                                         /* driver loading. */
};
typedef struct USBDriverDescription USBDriverDescription;

```

Field descriptions

`usbDriverDescSignature`

Set to `kTheUSBDriverDescriptionSignature` to indicate that the CFM library is a USB class driver.

`usbDriverDescVersion`

The structure version field. As of the initial release, this field should be set to `kInitialUSBDriverDescriptor`. For future releases of Mac OS USB, you may need to set this field to indicate support for new functionality.

The remaining fields of the driver description record are used to match the class driver to a device or interface. For most of the fields, a value of zero indicates that the field is not to be used to match the class driver to a device. For other fields such as the `usbDeviceClass` and `usbInterfaceClass`, a device and interface specific value must be entered.

`USBDeviceInfo`

The `usbVendorID` and `usbProductID` in the `USBDeviceInfo` structure are used to match a class driver to support both device drivers and interfaces. As indicated above for device and interface matching, the highest match value is generated when the `VendorID` and `ProductID` match along with the configuration value.

The `USBDeviceInfo` structure is defined as follows:

```

struct USBDeviceInfo {
    UInt16  usbVendorID;           /* USB Vendor ID */
    UInt16  usbProductID;         /* USB Product ID */
};

```

Writing Mac OS USB Drivers

```

    UInt16  usbDeviceReleaseNumber;    /* Release Number of Device */
    UInt16  usbDeviceProtocol;        /* Protocol Info. */
};

```

For the `USBDeviceInfo` fields, `VendorID` and `ProductID`, enter the values that correspond to the device that the class driver supports. For the `DeviceReleaseNumber` field, enter a BCD value for the device version. For the `usbDeviceProtocol` field, enter the protocol value assigned by the USB specification for your device. A value of 0 indicates that no device specific protocol is used and is a valid setting for the device. A value of 0xFF indicates that a vendor specific protocol is used. A match of the `VendorID` and the `ProductID` is the highest possible match.

`USBInterfaceInfo`

The information presented in the `USBInterfaceInfo` structure is used to match a class driver to support a device interface. When a device is detected as a composite class device, the composite class driver is loaded and tries to match class a class driver for each interface. For interface matching, the second highest match value is generated when the interface class is vendor specific AND the `VendorID` and interface subclass match. In case of duplicate matches, matching the interface protocol field for a driver raises its match level.

The `kUSBDoNotMatchInterface` bit must be clear in the `USBDriverLoadingOptions` field in order for the class driver to be checked to see if it can support an interface. For an interface match, the interface number must also match.

A class driver can support an interface class generically by not setting the `kUSBDoNotMatchGenericDevice` bit in the `USBDriverLoadingOptions` field. When this bit is clear, USB checks the class driver's support for the interface class, subclass and protocol against that for the device. When all three values match, the third highest match level is set for the driver. If the class and subclass match and the `kUSBProtocolMustMatch` bit is clear, then the fourth highest match level is set for the driver.

The `USBInterfaceInfo` structure is defined as follows:

Writing Mac OS USB Drivers

```

struct USBInterfaceInfo {
    UInt8  usbConfigValue;          /* Configuration Value */
    UInt8  usbInterfaceNum;        /* Interface Number */
    UInt8  usbInterfaceClass;      /* Interface Class */
    UInt8  usbInterfaceSubClass;   /* Interface SubClass */
    UInt8  usbInterfaceProtocol;   /* Interface Protocol */
};

```

USBDriverType

The information provided in the `USBDriverType` structure is used to match a class driver to support a device. For device matching, the second highest match level occurs when the device class is vendor specific, `0xFF`, and the `vendorID` in the `USBDeviceInfo` field, the device subclass and protocol fields all match.

A class driver can support a device class generically by not setting the `kUSBDoNotMatchGenericDevice` bit in the `USBDriverLoadingOptions` field. When this bit is clear, USB checks the class driver's support for the device class, subclass and protocol against that for the device. When all three values match, the third highest match level is set for the driver. If the class and subclass match and the `kUSBProtocolMustMatch` bit is clear, then the fourth highest match level is set for the driver.

The `USBDriverType` structure is defined as follows:

```

struct USBDriverType {
    Str31      nameInfoStr;        /* Driver's name when loading */
                                   /* into the Name Registry */
    UInt8      usbDriverClass;     /* USB Class this driver */
                                   /* belongs to */
    UInt8      usbDriverSubClass;  /* Module type */
    NumVersion usbDriverVersion;   /* Class driver version number */
};

```

The `nameInfoStr` field, is used by the USB Expert to record debugging information. The field is a Pascal string with the string length in byte 0. The `usbDriverClass` and `usbDriverSubClass` fields are used as described above to match a class driver generically to a device. Starting with version 1.2 of the Mac OS USB software, the `usbDriverVersion` field is used to distinguish class drivers.

USBDriverLoadingOptions

The `USBDriverLoadingOptions` field is used to control whether a class driver supports generic devices, interfaces, and other matching options. As of the v1.0 release, the following options are defined.

```
kUSBDoNotMatchGenericDevice = 0x00000001
```

The `kUSBDoNotMatchGenericDevice` bit indicates whether the class driver supports or does not support all devices or interfaces. When set, the class driver can only support a device with a matching `vendorID` as specified in the `USBDeviceInfo` structure. If this bit setting is clear, then the class driver module can support a device generically as specified in the section “USB Device and Driver Matching” (page 4-54).

```
kUSBDoNotMatchInterface = 0x00000002
```

The `kUSBDoNotMatchInterface` bit indicates whether the class driver supports USB interfaces. When set, the Mac OS USB software does not consider the driver for support of an interface. If this bit setting is clear, then the class driver module indicates that it can support an interface as defined in the settings of the `USBInterfaceInfo` structure.

```
kUSBProtocolMustMatch = 0x00000004
```

The `kUSBProtocolMustMatch` bit indicates whether a generic class driver can match when the class and subclass values match, but the protocol does not match. Set this bit in to require that USB match the protocol field, in addition to the class and subclass values during driver matching. Otherwise, leave this bit clear and a match value is generated when the class and subclass match.

The following example of a `USBDriverDescriptor` structure is for a vendor specific device.

```
USBDriverDescription TheUSBDriverDescription = {
    // Signature info
    kTheUSBDriverDescriptionSignature, // specifies USB Class Device Driver
    kInitialUSBDriverDescriptor,      // specifies the USB Class Driver version

    // Device Info
    kMyVendorID,                      // vendor, 0 = unspecified
```

Writing Mac OS USB Drivers

```

kMyProductID,           // product, 0 = unspecified
0,                      // version of product, 0 = unspecified
0,                      // protocol, 0 = not device specific

    // Interface Info
0,                      // Configuration Value, 0 = unspecified
0,                      // Interface Number, 0 - interface 0
kUSBVendor_Specific,   // Interface Class, 0x03 - HID Interface
0,                      // Interface SubClass
0,                      // Interface Protocol

    // Driver Info
"\pUSBCustomDriver",   // Driver name for Name Registry
kUSBVendor_Specific,   // Device Class, 0x03 HID Interface class
0,                      // Device Subclass
kMyHexMajorVers,
kMyHexMinorVers,
kMyCurrentRelease,
kMyReleaseStage,       // version of driver

    // Driver Loading Info
kUSBDoNotMatchGeneric  // Flags 2 = don't care if protocol matches
                        // will not match a generic device and interface
                        // will match to an interface

```

USBClassDriverPluginDispatchTable Structure

The `USBClassDriverPluginDispatchTable` is the second exported symbol that the CFM class driver module must export. The Mac OS USB software looks for this symbol in order to initialize the class driver module. The dispatch table structure contains pointers to functions that your driver must support to operate properly in the Mac OS USB environment.

Typically, class drivers may require additional exports to facilitate communication with a shim of other Mac OS service.

The structure of `USBClassDriverPluginDispatchTable` for class drivers is as follows:

```

struct USBClassDriverPluginDispatchTable {
    UInt32                pluginVersion;
    USBDValidateHWProcPtr validateHWProc;    /* Proc for driver to */
}

```

Writing Mac OS USB Drivers

```

                                /* verify proper HW */
USBDDInitializeDeviceProcPtr    initializeDeviceProc; /* Proc that initializes */
                                /* the class driver */
USBDDInitializeInterfaceProcPtr initializeInterfaceProc; /* Proc that */
                                /* initializes particular */
                                /* interface in the */
                                /* class driver */
USBDDFinalizeProcPtr           finalizeProc; /* Proc that finalizes */
                                /* the class driver */
USBDDDriverNotifyProcPtr      notificationProc; /* Proc to pass */
                                /* notifications to */
                                /* the driver */
};

```

You must initialize the `pluginVersion` field to `kClassDriverPluginVersion` defined in the header file `USB.h`. The USB Manager uses this field to distinguish between future versions of the dispatch table.

The following fields of the dispatch table are required when the device is initialized

- `validateHWProc`
- `intializeDeviceProc`
- `finalizeProc`

If any of these fields are `nil`, then the `kUSBBadDispatchTable` error is returned to the USB Family Expert, and the driver load fails. Each of these calls is made at task time, and each is made synchronously.

You are advised to install a `notificationProc`, so that you can handle a "hot unplug" situation. However, if this field is `nil`, the device driver load still continues.

If your class driver can be used to communicate with a specific interface portion of a composite device, the following fields of the dispatch table are required.

- `initializeInterfaceProc`
- `finalizeProc`

If either of these fields are `nil`, then the `kUSBBadDispatchTable` error is returned, and the interface load fails. Each of these calls are made at task time, and each is made synchronously.

Writing Mac OS USB Drivers

As above, you are advised to install a `notificationProc`, so that you can handle a "hot unplug" situation. However, if this field is `nil`, the interface driver load still continues.

The contents of the fields in the `USBClassDriverPluginDispatchTable` are defined in the following sections.

ValidateHWProc Function

The `validateHWProc` function is called for a USB class driver to validate that the device is supported by the selected driver. The prototype for the `validateHWProc` is

```
OSStatus USBValidateHWProcPtr(
    USBDeviceRef device,
    USBDeviceDescriptorPtr pDesc);
```

The USB Manager finds the best matching class driver for a device, then calls the `USBValidateHWProcProcPtr` to provide the class driver with a chance to check the device descriptor itself to verify that the device is supported. This procedure call is made at system task time, and is issued synchronously. The class driver may want to verify that a certain minimum device hardware version exists, or lock out support for specific incompatible products or versions.

The driver returns a `kUSBNoErr OSStatus` result, if the class driver supports the hardware, or a value other than `kUSBNoErr` to indicate that the driver does not support the hardware. The `validateHWProc` should be used solely to validate the support for a specific hardware device. It should not be used to set up the driver connection. To set up the driver connection use the `initializeDeviceProc` function, which is discussed next.

InitializeDeviceProc Function

The Mac OS USB software calls the `initializeDeviceProc` function to tell the class driver to prepare for communications with the USB device. The prototype for the `initializeDeviceProc` function is

```
OSStatus USBInitializeDeviceProcPtr(
    USBDeviceRef device,
    USBDeviceDescriptorPtr pDesc,
    UInt32 busPowerAvailable);
```

The `initializeDeviceProc` function is called at system task time. The class driver may wish to allocate memory, set up resources (see the “Mac OS USB Compatibility With Mac OS Toolbox Calls” section), and initiate the necessary sequence of USB calls to configure the device. This includes identifying the available configurations, specifying the configuration and interface to be used, opening endpoints and pipes, specifying the protocol to be used, installing an interrupt routine, and related actions.

The description that follows applies for cases when the class driver is called through either the `initializeDeviceProc` or the `initializeInterfaceProc` function. Both of these functions are called synchronously and at system task time. If either the `initializeDeviceProc` or `initializeInterfaceProc` functions return anything other than `kUSBNoErr`, the driver is not loaded, and the USB Manager attempts to load the next-best matching driver. Before returning from the function, the driver must complete any calls that must be made at task time. After doing so, the driver should initiate an asynchronous “state machine process” to complete the connection. This can include memory allocation, assuming that the `USBAllocMem` call is made. Once the asynchronous state machine process is begun, the driver can return from the `initializeDeviceProc` function call.

As a reminder, most USB calls are asynchronous. As stated in “Polling Versus Asynchronous Completion (Important)” (page 5-103), a USB driver can NOT poll for completion of these asynchronous USB calls by simply checking the `usbStatus` field without using a completion routine.

All of the USB example class drivers found in the Mac OS USB DDK implement an asynchronous state machine startup process. In the driver examples, the last thing that the `initializeDeviceProc` or `initializeInterfaceProc` code does before returning to the caller, is to make an `initiateTransaction` call. The `initiateTransaction` call uses the `usbRefCon` field in the USB parameter block, defined in “The USBPB Parameter Block” (page 5-94), as a selector into a switch statement. After the first call to `initiateTransaction` is initiated, control returns to the caller. As the asynchronous call completes, the completion routine checks the call results, modifies the `usbRefCon` field appropriately and issues a call to the `initiateTransaction` function. The `initiateTransaction` function checks the `usbRefCon` field and issues a new asynchronous call to handle the next step in the state machine. This continues until all of the processing is complete.

InitializeInterfaceProc Function

The Mac OS USB software calls the `initializeInterfaceProc` function to tell the class driver to prepare for communications with the USB interface. The prototype for the `initializeInterfaceProc` function is

```
OSStatus USBDInitializeInterfaceProcPtr(
    UInt32 interfaceNum,
    USBInterfaceDescriptorPtr pInterface,
    USBDeviceDescriptorPtr pDevice,
    USBInterfaceRef interfaceRef);
```

A composite USB device generally results in the load of the composite class driver. The composite class driver obtains information about the first configuration, verifies that the bus power required for the device is supported by the hub, then sets the configuration. A new interface reference is created for each interface in the selected configuration, and the USB Manager attempts to find and load a driver for that interface. The interface matching process is described in “USB Device and Driver Matching” (page 4-54).

The `initializeInterfaceProc` function is called synchronously at system task time. If the `initializeInterfaceProc` function returns anything other than `kUSBNoErr`, the driver is not loaded, and the USB Manager attempts to load the next-best matching driver. The class driver may wish to allocate memory, set up resources (see “Mac OS USB Compatibility With Mac OS Toolbox Calls”), and initiate the necessary sequence of USB calls to configure the interface. This includes specifying the interface to be used, opening endpoints and pipes, specifying the protocol to be used, installing an interrupt routine, and related actions.

Before returning from the function, the driver must complete any calls that must be made at task time. After doing so, the driver should initiate an asynchronous “state machine process” to complete the connection. This can include memory allocation, assuming that the `USBAllocMem` call is made. Once the asynchronous state machine process is begun, the driver can return from the `initializeInterfaceProc` function call.

All of the USB example class drivers found in the Mac OS USB DDK implement an asynchronous state machine startup process. In the examples, the last thing that the `initializeInterfaceProc` code does before returning to the caller, is to make an `InitiateTransaction` call. The `initiateTransaction` call uses the `usbRefCon` field as a selector into a switch statement. After the first call to `initiateTransaction` is initiated, control returns to the caller. As the

asynchronous call completes, the completion routine checks the call results, modifies the `usbRefCon` field appropriately and issues a call to the `initiateTransaction` function. The `initiateTransaction` function checks the `usbRefCon` field and issues a new asynchronous call to handle the next step in the state machine. This continues until all of the processing is complete.

Driver notificationProc Function

The driver `notificationProc` function receives device notification messages from the Mac OS USB software. The prototype for the `notificationProc` function is

```
OSStatus USBDDriverNotifyProcPtr(
    USBDriverNotification notification,
    void *pointer,
    UInt32 refcon);
```

The Mac OS USB software calls the driver's `notificationProc` function with a message when something the driver needs to be aware of is happening. There are many messages passed both internally to the USB software and to drivers to communicate what is happening on the USB. The notification messages that are of importance to class drivers are:

```
kNotifySystemSleepRequest = 0x00000001,
kNotifySystemSleepDemand = 0x00000002,
kNotifySystemSleepRevoke = 0x00000003
kNotifyHubEnumQuery = 0x00000006,
kNotifyChildMessage = 0x00000007,
kNotifyExpertTerminating = 0x00000008,
kNotifyDriverBeingRemoved = 0x0000000B
```

The sleep notification messages `kNotifySystemSleepRequest` and `kNotifySystemSleepDemand` are the same on any power-managed system. On PowerBook models, the processor and I/O subsystems are turned off when the machine goes to sleep. Because the USB is part of the I/O subsystem on PowerBooks with built-in USB or USB on a PC Card, the sleep state does effect USB drivers. Desktop Macintosh computers do not see these messages, since the processor and network I/O subsystems on desktop models remain active during system sleep.

When a PowerBook computer with Mac OS USB software version 1.2 or later goes to sleep, the USB Manager sends any active USB drivers the `kNotifySystemSleepDemand` message and then unloads the driver just like a

disconnect (hot unplug). When the PowerBook wakes up, the USB software re-enumerates the USB and appropriate drivers are loaded for any USB devices found on the bus.

FinalizeProc Function

The Mac OS USB software calls the `finalizeProc` function just before unloading the driver. This function is not called if any of the `validate` or `initialize` procs have returned an error. The prototype for the `finalizeProc` function is

```
OSStatus USBDFinalizeProcPtr(
    USBDeviceRef device,
    USBDeviceDescriptorPtr pDesc);
```

A `kNotifyDriverBeingRemoved` notification is sent prior to the `finalizeProc` actually being called. When this notification is received by the driver's `notificationProc` function, it is the last chance a driver has to run any cleanup or user notification code before being unloaded.

The `finalizeProc` function is called at system task time and is called synchronously. When the `finalizeProc` function returns, the class driver may be unloaded from memory. A crash can occur if the `finalizeProc` function is called while the driver has USB calls pending completion. The crash occurs because immediately after returning from the `finalizeProc` function, the class driver is unloaded from memory and any outstanding completion procs may become invalid. To protect against this problem and get additional details about the use of the `notificationProc` function, refer to the section on "Handling Hot Unplugging, Dealing With Notifications"

Handling Hot Unplugging, Dealing With Notifications

A class driver module must be designed to handle a "hot unplug" situation, in which the client process still has the class driver open. When the Mac OS USB software detects that a device has been unplugged, but before the driver `finalizeProc` is called, the Mac OS USB software calls at system task time, the `notificationProc` that was registered in the driver's `USBClassDriverPluginDispatchTable`. The `notificationProc` is sent the `kNotifyDriverBeingRemoved` message. If the driver still has a client connection to maintain, then it can return the `kUSBDeviceBusy` result. By returning the

Writing Mac OS USB Drivers

`kUSBDeviceBusy` result, the Mac OS USB software holds off calling the driver's `finalizeProc` function. The Mac OS USB software then periodically calls the `notificationProc` with the `kNotifyDriverBeingRemoved` message until the `kUSBNoErr` result is returned. Once the `kUSBNoErr` returns, the driver's `finalizeProc` is called, after which the driver code fragment is unloaded from memory.

Using the mechanism described above provides a means to protect your driver from being unloaded after the device has been removed by a hot unplug. By continuing to return the `kUSBDeviceBusy` result, you can hold off execution of the `finalizeProc`. Note that any calls into the USL to your device, will fail.

This raises the issue as to what happens to pending calls, when the device is unplugged. These calls may complete with error `kUSBNotRespondingError`, or they may be aborted by the pipe closure process with error `kUSBAbortedError`. Note that the class driver should never retry an unexpected `kUSBAbortedError` in the completion routine. The order of these errors or notifications is not guaranteed. They may occur after the `notificationProc` function is called with the `kNotifyDriverBeingRemoved` notification message. If this event occurs, then make the `USBAbortPipeByReference` function call for each pipe with active transactions. The `notificationProc` must wait until all of the transactions have completed, before returning `kUSBNoErr` in response to the `kNotifyDriverBeingRemoved` message. If the `kUSBNoErr` response is returned with pending transactions incomplete, then the driver's `finalizeProc` is called, the class driver is unloaded, and the system crashes when the completion routine for the transaction finally completes (for example, with a `kUSBAbortedError` status) at some point later on.

Communicating With Client Processes

To this point, the discussion in this chapter has been about how a Mac OS USB driver communicates with a USB device. Ultimately, there must be communication that takes place between the USB class driver and the Mac OS applications and processes. While there are specific guidelines for how USB drivers and devices communicate with each other, there is no official API for how client processes communicate with USB drivers. This section discusses different strategies for facilitating these communications.

The Disappearing Driver

The USB class driver is designed to be loaded and present to support an attached USB device. When the device is detached, the class driver is unloaded from memory. This makes the USB class driver model unsuitable for client processes that are not prepared to deal with a disappearing driver.

Common Ground and The Compatibility Shim

For most device I/O, a device driver is implemented to present a standard I/O API to client processes. For serial communications and disk storage devices, this may mean that a `DRVR/NDRV` style resource is loaded into memory with a corresponding unit entry in the device table. Keyboards, mice, and graphics tablets typically register a driver with the Cursor Device Manager. A digital camera implements a video digitizer (also known as a `vdig`). A printer requires a print chooser device. The key thing to understand is how existing non-USB devices, which are similar to your USB device, currently communicate with Mac OS clients. Once you understand how the non-USB device model works, you then develop a similar type of device driver, one that presents a familiar API to the client process, but also knows how to communicate with the USB class driver. For purposes of this discussion, this modified driver is called a *compatibility shim*.

Where To Implement a Compatibility Shim

It's important to understand that the compatibility shim cannot be implemented within the class driver. As mentioned previously, if a device is unplugged suddenly, the class driver is called via its `finalizeProc` function. Upon completion of the `finalizeProc`, the shared library driver code may be unloaded from memory. If the compatibility shim is included in the class driver, then the shim code also disappears.

In addition, with Mac OS X, it's important to isolate those portions of code that require change. This is especially true of the compatibility shim, which may require an API under the current OS that may not be supported in the future. For this reason, Apple advises that you not implement the compatibility shim within your class driver code.

Designing A Compatibility Shim

In considering the design of a compatibility shim, you should be familiar with the method by which application processes communicate with similar non-USB devices. In many cases, the compatibility shim should be thought of as a resident process that client processes can easily find. When the client process opens the compatibility shim, the shim determines whether the USB device is attached and initiates communications with the class driver. The class driver knows about making calls to the USB device. The compatibility shim knows about communicating with the Mac OS services.

Note

If you are designing a DRVR style driver, you may find the `TradDriverLoadLibrary` code invaluable. This code is available on the Developer CD, Tool Chest volume. The Developer CD also includes a `RAMDisk` sample that demonstrates an implementation of a working DRVR resource.

At some point, the compatibility shim finds the class driver, and implements communications between the class driver, the Mac OS, and finally the user. A discussion of the means by which the compatibility shim code can detect the presence of the class driver is provided in “Detecting USB Device Presence” (page 4-83).

One method for facilitating communication between the Mac OS and the device is for the class driver to export a dispatch table. The `USBClassDriverPluginDispatchTable` is an example of this mechanism. The USB Manager imports the `USBClassDriverPluginDispatchTable` and calls the functions defined in the dispatch table as necessary. Your class driver could define such a dispatch table and export the dispatch table symbol. The compatibility shim, would use the `FindSym` call to obtain the address of the dispatch table, then make calls to the functions that are defined in the dispatch table as appropriate.

For some USB devices, there is no standard mechanism for communications with client processes. It is possible for an application to do exactly what the compatibility shim does. If an application does communicate directly with the USB class driver, an important consideration concerns the hot-unplug situation. The application must be ready to handle this situation, just as the compatibility shim must be designed to do so also. The application should also provide support for multiple devices and dynamically allow them to be used without requiring the user to quit and relaunch the application.

Helpful Resources For Compatibility Shim Development

This section points you to sample code which you may find helpful in designing compatibility shim code.

For many compatibility shims, a 68K DRVR code resource may prove useful to implement. While this is legacy code and is not supported under Mac OS X, it is the required mechanism for device Input/Output for many client processes.

For a USB Communications Class devices, the compatibility shim must provide a 68K driver and also register the DRVR with the Communications Toolbox (CRM) Comm Resource Manager. This allows the USB Communications device to be recognized by all CRM aware applications. For more information on the 68K DRVR mechanism, refer to *Inside Macintosh: Devices*. A description of the standard process by which Mac OS applications find and open serial devices, is presented in Tech Note 1119, *Serial Port Apocrypha*.

For a sample of implementing a 68K 'DRVR' code resource, there is the RAMDisk v1.45 code sample. This sample is also useful in demonstrating how to set up a Metrowerks CodeWarrior project to create a DRVR code resource. The sample code demonstrates the use of the TradDriverLoaderLib code that facilitates the installation of all DRVR code resources into the Device Manager Device Unit Table.

For Video Input Devices, such as a USB camera, the standard mechanism for processing data from hardware, is the Video Digitizer. You can learn more Video Digitizers in *Inside Macintosh: QuickTime Components*, available from the QuickTime Developer Documentation Web Page.

For Networking Devices, such as an Ethernet/USB convertor, an Open Transport DLPI Driver must be implemented. A sample Loopback DLPI driver is provided as part of the Open Transport SDK. You can obtain the latest Open Transport SDK from the Apple Developer Open Transport Web Page.

For Print class devices, a compatibility shim would provide a Chooser Printer Driver. An example Chooser Printer Driver is provided with the USB DDK, in the Examples folder.

Communicating with the Human Interface Device Driver

For HID (Human Interface Devices), there can be a number of different shims such as the Input Sprocket (used by many games to handle a wide range of controllers). The shims communicate with the HID driver through a dispatch table. Originally, versions of the HID class driver provided by the USB DDK

contained `TheHIDModuleDispatchTable`. Starting with version 1.3.7 of the USB software, a more flexible dispatch table, `TheHIDDeviceDispatchTable`, has been used. For backward compatibility, the HID class driver provides both dispatch tables. However, to insure your software works with future USB HID devices, the new `TheHIDDeviceDispatchTable` should be used.

The HIDDeviceDispatchTable Structure

The `HIDDeviceDispatchTable` is defined as follows:

```
struct HIDDeviceDispatchTable {
    UInt32                dispatchTableCurrentVersion;
    UInt32                dispatchTableOldestVersion;
    UInt32                vendorID;
    UInt32                vendorSpecific;
    UInt32                reserved;
    HIDGetDeviceInfoProcPtr  pHIDGetDeviceInfo;
    HIDGetHIDDescriptorProcPtr  pHIDGetHIDDescriptor;
    HIDOpenDeviceProcPtr    pHIDOpenDevice;
    HIDCloseDeviceProcPtr   pHIDCloseDevice;
    HIDInstallReportHandlerProcPtr  pHIDInstallReportHandler;
    HIDRemoveReportHandlerProcPtr  pHIDRemoveReportHandler;
    HIDCallPreviousReportHandlerProcPtr  pHIDCallPreviousReportHandler;
    HIDGetReportProcPtr     pHIDGetReport;
    HIDSetReportProcPtr     pHIDSetReport;
    HIDControlDeviceProcPtr  pHIDControlDevice;
};
```

The `dispatchTableCurrentVersion` field identifies which version of driver this dispatch table supports. The normal value to assign here is `kHIDCurrentDispatchTableVersion`. As of USB DDK 1.4, the HID class driver has implemented `pHIDGetReport` and `pHIDSetReport`, which bumped its version to 2.

The `dispatchTableOldestVersion` field is set to the version of the oldest built client using this API that this HID device driver works with. The normal value to assign here is `kHIDOldestCompatibleDispatchTableVersion`. While new functions were added for version 2, the structure of the dispatch table remains the same as version 1, so that is the value of `kHIDOldestCompatibleDispatchTableVersion` provided with USB DDK 1.4.

Writing Mac OS USB Drivers

The `vendorID` field identifies who wrote this HID device driver. Valid data here would be a high word of 0 and the low word would contain the USB vendorID.

The `vendorSpecific` field is a way to pass specific flags or selectors to a driver that is not required by generic HID drivers, such as the HID class driver.

The `HIDGetDeviceInfoProcPtr` is used to get specific information about a HID device. The prototype for this function is defined as follows:

```
OSStatus HIDGetDeviceInfoProcPtr (
    UInt32 inInfoSelector,
    void * outInfo
    UInt32 * ioSize);
```

The selectors for the type of information that can be gotten from the device are:

```
kHIDGetInfo_VendorID
kHIDGetInfo_ProductID
kHIDGetInfo_VersionNumber
kHIDGetInfo_MaxReportSize
kHIDGetInfo_GetManufacturerString
kHIDGetInfo_GetProductString
kHIDGetInfo_GetSerialNumberString
kHIDGetInfo_GetIndexedString
```

The `HIDGetHIDDescriptorProcPtr` is used to get the HID descriptor in its native, unparsed form from the device. This descriptor must be used by the HID Library to interpret reports issued by the HID class driver. To get the HID descriptor, the caller must specify the `inDescriptorType` to be `kUSBReportDesc` and index of 0. The caller also allocates the buffer for the HID descriptor. Calling this with a buffer size of 0 returns the required size in the `ioBufferSize` variable. The prototype for this function is defined as follows:

```
OSStatus HIDGetHIDDescriptorProcPtr (
    UInt32 inDescriptorType,
    UInt32 inDescriptorIndex,
    void * outDescriptor
    UInt32 * ioBufferSize);
```

Writing Mac OS USB Drivers

The `HIDOpenDeviceProcPtr` is used to open a HID device before the HID Library can use it. The prototype for this function is defined as follows:

```
OSStatus HIDOpenDeviceProcPtr (
    HIDDeviceConnectionRef outConnectionRef,
    UInt32 permissions,
    UInt32 reserved);
```

The `HIDOpenDeviceProcPtr` must also specify what type of access the shim wants of the HID device in the `permissions` parameter. Possible permissions are:

```
kHIDPerm_ReadOnly
kHIDPerm_ReadWriteShared
kHIDPerm_ReadWriteExclusive
```

The `HIDCloseDeviceProcPtr` is used to close the HID device when the HID Library has finished using it. The prototype for this function is defined as follows:

```
OSStatus HIDCloseDeviceProcPtr (
    HIDDeviceConnectionRef inConnectionRef);
```

IMPORTANT

The dispatch table entries that follow rely upon a report handler. This report handler is a user provided function to be called when a HID report is received. When a report is passed in, the HID driver manages the memory provided for the report. The report handler is of type `HIDReportHandlerProcPtr` and is defined as follows:

```
void HIDReportHandlerProcPtr (
    void * inHIDReport,
    UInt32 inHIDReportLength,
    UInt32 inRefcon);
```

The `HIDInstallReportHandlerProcPtr` is used to install a HID report handler. When installed, a report handler is called at deferred task time, unless `kHIDFlag_CallbackIsResident` is passed. Whenever an interrupt packet is

Writing Mac OS USB Drivers

received on the first interrupt in pipe, the supplied refcon is also passed through. The prototype for this function is defined as follows:

```
OSStatus HIDInstallReportHandlerProcPtr (
    HIDDeviceConnectionRef inConnectionRef,
    UInt32 flags,
    HIDReportHandlerProcPtr inReportHandlerProc,
    UInt32 inRefcon);
```

The `HIDRemoveReportHandlerProcPtr` is used to remove a report handler when it is no longer needed. When the current report handler is removed, any previously installed report handler that is still valid is restored to handle the HID reports. The prototype for this function is defined as follows:

```
OSStatus HIDRemoveReportHandlerProcPtr (
    HIDDeviceConnectionRef inConnectionRef);
```

The `HIDCallPreviousReportHandlerProcPtr` can be used within your report handler. If there was a `HIDReportHandler` installed previous to the one from this connection, use this proc ptr to pass the report on to that previous handler. The prototype for this function is defined as follows:

```
OSStatus HIDCallPreviousReportHandlerProcPtr (
    HIDDeviceConnectionRef inConnectionRef,
    void * inHIDReport,
    UInt32 inHIDReportLength);
```

The `HIDGetReportProcPtr` is used to get a report of type input, output, or feature, directly to a HID device. The HID class driver buffers the information and returns immediately to the caller. Then at a safe time, the driver passes this command on to the device. In the event that the command buffers are full, `HIDGetReportProcPtr` returns `memFullErr` and you will have to reissue the command at a later time. When HID driver finally gets the information from the device, the report is returned directly to the `HIDReportHandlerProcPtr`. The `HIDGetReportProcPtr` was not available until version 2 of the dispatch table and was listed as reserved in version 1. The HID class driver did not publicly implement this function until USB DDK 1.4. Therefore, check for a nil value

Writing Mac OS USB Drivers

before dispatching through this proc pointer. The prototype for this function is defined as follows:

```
OSStatus HIDGetReportProcPtr (
    HIDDeviceConnectionRef inConnectionRef,
    UInt32 inReportType,
    UInt32 inReportID,
    HIDReportHandlerProcPtr inReportHandlerProc,
    UInt32 inRefcon);
```

The `HIDSetReportProcPtr` is used to send a report of type input, output, or feature, directly to a HID device. Before calling this function, the calling routine must allocate memory for the report and use the HID Library functions, documented in chapter 7, to encode the values. The HID class driver copies the required information, so this memory can be deallocated upon return from this call. This proc pointer was not available until version 2 of the dispatch table and was listed as reserved in version 1. The HID class driver did not publicly implement this function until USB DDK 1.4. Therefore check for a nil value before dispatching through this proc pointer. Note that when this call completes, there may be no notification to the calling code that this set report has succeeded. It may be necessary to later call `HIDGetReportProcPtr` to get confirmation. The prototype for `HIDSetReportProcPtr` is defined as follows:

```
OSStatus HIDSetReportProcPtr (
    HIDDeviceConnectionRef inConnectionRef,
    UInt32 inReportType,
    UInt32 inReportID,
    void * inInfo,
    ByteCount inSize);
```

The `HIDControlDeviceProcPtr` is used to send miscellaneous control messages to the device. The prototype for this function is defined as follows:

```
OSStatus HIDControlDeviceProcPtr (
    HIDDeviceConnectionRef inConnectionRef,
    UInt32 inControlSelector,
    void * ioControlData);
```

The only selector currently available for the `inControlSelector` parameter is:

```
KHIDVendorSpecificControlStart
```

The USBHIDModuleDispatchTable Structure

This section provides an example of how a cursor device shim and an ADB shim communicate with a class driver.

Note

The information provided in this section describes an early version of the HID dispatch table, and is documented here for backward compatibility. The information is still valid with respect to shim and driver communication. However, while the `HIDDeviceDispatchTable` documented above is similar to the `USBHIDModuleDispatchTable` documented here, the newer `HIDDeviceDispatchTable` differs in that it allows much more flexibility in requesting reports, getting all the data the device is capable of supplying, and sending reports to the device. If your device is a HID device, it is suggested you write a client for the HID class driver using the `HIDDeviceDispatchTable` and the HID Library APIs documented in Chapter 7, “HID Library Reference.”

The dispatch table is the `USBHIDModuleDispatchTable` and is defined as follows:

```
struct USBHIDModuleDispatchTable {
    UInt32                hidDispatchVersion;
    USBHIDInstallInterruptProcPtr  pUSBHIDInstallInterrupt;
    USBHIDPollDeviceProcPtr    pUSBHIDPollDevice;
    USBHIDControlDeviceProcPtr  pUSBHIDControlDevice;
    USBHIDGetDeviceInfoProcPtr  pUSBHIDGetDeviceInfo;
    USBHIDEnterPolledModeProcPtr pUSBHIDEnterPolledMode;
    USBHIDExitPolledModeProcPtr pUSBHIDExitPolledMode;
};
```

The `USBHIDInstallInterruptProcPtr` is used to install the interrupt routine that is called to process incoming data. The prototype for this function is defined as follows:

```
OSStatus USBHIDInstallInterruptProcPtr(
    HIDInterruptProcPtr HIDInterruptFunction,
    UInt32 refcon);
```

When the class driver is called via this dispatch proc pointer, the class driver must save the `HIDInterruptProcPtr` and the `refcon`. When incoming data has

Writing Mac OS USB Drivers

been received, call the registered `HIDInterruptProcPtr` passing the saved refcon value as the first parameter, and the appropriate data structure pointer, as the second parameter. For keyboard devices, this is the `USBKeyboardData` structure. For mouse devices, this is the `USBMouseData` structure. The prototype for the `HIDInterruptProcPtr` function is defined as follows

```
void HIDInterruptProcPtr(UInt32 refcon, void *theData);
```

The `USBHIDPollDeviceProcPtr` is used to ?????

The `USBHIDControlDeviceProcPtr` is used to have the device perform device specific functions. The prototype for this function is defined as follows:

```
OSStatus USBHIDControlDeviceProcPtr(
    UInt32 theControlSelector,
    void *theControlData);
```

The selector values which can be passed depend on the type of device. For mouse type HID devices, the selector values that can be expected are as follows

`kHIDRemoveInterruptHandler` - clear out the `ShimInterruptHandler`, as well as the associated refcon and save interrupt handler

`kHIDEnableDemoMode` - save the current interrupt handler, set up a `Cursor Device Manager` device, and install a data processing routine similar to the `USBMouseIn` routine from the file `HIDEmulation.c` from the `Mouse Module` example class driver. Refer to the `USB DDK` for this example.

`kHIDDisableDemoMode` - this call is made following a control call to `kHIDEnableDemoMode`. Dispose of the current `Cursor Device Manager` structure and replace the interrupt handler with the saved interrupt handler.

For keyboard type HID devices, the selector values that can be expected are as follows:

`kHIDSetLEDStateByBits` - command issued to device to set state of the keyboard LEDs. Refer to the `KBDHIDEmulation.c` file in the `DDK` for code that shows how this call should be handled.

`kHIDRemoveInterruptHandler` - clear out the current interrupt handler along with the refcon and any saved interrupt handler information.

`kHIDEnableDemoMode` - save the current interrupt routine and replace it with a version like the `USBDemoKeyIn` code. This routine is

The `USBHIDGetDeviceInfoProcPtr` is used to (to be completed later)

The `USBHIDEnterPolledModeProcPtr` is used to (to be completed later)

The `USBHIDExitPolledModeProcPtr` is used to (to be completed later)

Detecting USB Device Presence

There are two different means for finding a USB device.

- Use the `USBGetNextDeviceByClass` call to check for the immediate presence of a device and locate its class driver.
- Use the `USBInstallDeviceNotification` mechanism to be alerted when a device or interface is added or removed.

Use the `USBGetNextDeviceByClass` call to determine the presence of a device. This call returns the `CFragConnectionID` associated with the class driver. Use the `CFragConnectionID` with the `FindSym` call to obtain the address of an exported symbol, for example the address to a proc pointer dispatch table. When making the `USBGetNextDeviceByClass`, you want to set the input `USBDeviceRef` parameter to `kNoDeviceRef` so that it finds the first instance of a device. Leave the `usbDeviceRef` value set to the last found device to find all devices of this device.

Listing 4-1 is an example of using the `USBGetNextDeviceByClass` function to look for a keyboard device. If the keyboard is found, the sample looks for the `TheUSBHIDModuleDispatchTable` export symbol. All HID class drivers are required to export a dispatch table symbol. In the case of older HID drivers, the symbol is `TheUSBHIDModuleDispatchTable`, current implementation should export the symbol `TheHIDDeviceDispatchTable`, which is documented in Chapter 7, “HID Library Reference.” See “The `USBHIDModuleDispatchTable` Structure” for additional information about the contents of the `USBHIDModuleDispatchTable` used in this discussion.

Listing 4-1 Detecting a keyboard device

```
UInt16  deviceClass = 0x0003;
UInt16  deviceSubClass = 0x0001;
UInt16  keyboardDeviceProtocol = 0x0001;

CFragConnectionID  keyboardConnID = kUSBNoDeviceRef;
CFragSymbolClass   symClass;
```

Writing Mac OS USB Drivers

```

THz                currentZone;
currentZone = GetZone ();
SetZone (SystemZone ()); /* Class drivers are always loaded in the System Zone */

while (USBGetNextDeviceByClass(&keyboardConnID, deviceClass,
                               deviceSubClass, keyboardDeviceProtocol))
{
    status = FindSymbol (keyboardConnID, "\pTheUSBHIDModuleDispatchTable",
                        (Ptr *)&TheKeyboardDispatchTable, &symClass);
}
SetZone (currentZone);

```

The significance of the sample in Listing 4-1, is that it demonstrates a way for the compatibility shim code to find a procedure pointer table defined by the class driver. This mechanism also demonstrates a limitation for writing universal handlers in that all other devices of the same type would need to implement the exact same procedure pointer tables and functionality.

For compatibility shims that need to handle hot plug-in device connections, there is a mechanism for notification of `AddDevice`, `AddInterface`, `RemoveDevice`, and `RemoveInterface` events. Use the `USBInstallDeviceNotification` function to install a notification handler.

The sample code in Listing 4-2 demonstrates the use of the `USBInstallDeviceNotification` call. Note that the sample is designed to have the notification routine called whenever any USB device is plugged in or unplugged from the USB chain.

Listing 4-2 Using the `USBInstallDeviceNotification` function

```

pb.usbDeviceNotification = -1;           // tell me about everything
pb.usbClass = kUSBHIDInterfaceClass;    // want to know about HID class devices
pb.usbSubClass = kUSBAnySubClass;       // tell me about all subclass devices
pb.usbProtocol = -1;                    // don't care about the protocol used by device
pb.usbVendor = -1;                       // allow any vendors keyboard notification
pb.usbProduct = -1;                      // allow any product ID notification
pb.result = noErr;
pb.callback = (USBDeviceNotificationCallbackProcPtr)&myNotificationCallback;
pb.refcon = nil;

USBInstallDeviceNotification (&pb);

```

Writing Mac OS USB Drivers

For the notification sample described above, the notification proc might be as shown below.

```
OSStatus ShimOpenDriver(USBDeviceRef theDevRef)
{
    OSStatus          theErr = 0;
    CFragSymbolClass  symClass;
    CFragConnectionID connID;
    THz               currentZone;

    theDevRef = 0;
    currentZone = GetZone();    // save the current zone setting to restore to later
    SetZone (SystemZone ());   // set the current zone to the system zone
                                // look for the desired device
    theErr = USBGetNextDeviceByClass(&theDevRef, &connID,
                                    kUSBInterestingClass, 0, 0);
    SetZone (currentZone);     // restore the zone

    if (theErr == noErr)
    {
        SetZone(SystemZone()); // set the current zone to the system zone
                                // find the desired exported symbol
        theErr = FindSymbol(connID, "\pTheUSBHIDModuleDispatchTable",
                            (Ptr *)&TheDispatchTable, &symClass);

        SetZone (currentZone); // restore the zone
    }
    return theErr;
}

void myNotificationCallback(USBDeviceNotificationParameterBlock *pb)
{
    switch(pb->usbDeviceNotification) // why were we notified?
    {
        case kNotifyAddInterface:     // because a HID device appeared
            ShimOpenDriver(pb->usbDeviceRef);
            break;

        case kNotifyRemoveInterface:  // because a HID device disappeared
            break;
    }
}
```

```

default:
    break;
}
}

```

The format of the exported symbol `TheUSBHIDModuleDispatchTable` referred to in the sample code above would be specific for use between the shim and class driver. Some elements of the dispatch table might include a notification proc pointer, and other proc pointers to handle read, write, control, status, and killio requests from the client.

Mac OS USB Compatibility With Mac OS Toolbox Calls

The use of Mac OS Toolbox calls is discouraged from within a USB class driver, however, there are cases when the Toolbox calls can be made. Carefully consider whether a Toolbox call is really required within the class driver, as opposed to implementing the functionality within the compatibility shim. There are cases where the use of Toolbox calls make sense to handle concerns that USB Manager may not be able to handle.

It is important to understand that the class driver typically operates at secondary interrupt time, under interrupt conditions. Many of the Toolbox calls cannot be made in an interrupt context. The exception to this guideline is that the `intializeProc` and the `finalizeProc` are called at system task time.

With regard to memory allocation, the use of USL function `USBA11ocMem` is preferred over the `NewPtrSys` call. The `USBA11ocMem` call is designed to use an appropriate memory allocation method for the system software releases. The `NewPtrSys` call may be supported under the current implementation of USB, but it may not be in the future. Note that one can make the `NewPtrSys` call during the `intializeProc`.

To handle preferences, the preferred solution is to have the compatibility shim access the Resource Manager, then tell the class driver to implement the setting via the `ControlProc`. The `ControlProc` would be a proc imported to the compatibility shim as described above.

The 1.2 and earlier releases of Mac OS USB do not provide timeout support for USB calls. Where this is a requirement, the class driver could implement a watchdog Time Manager task, to check if a USB call has completed in a

specified period of time. Examples of using the Time Manager can be found in “Inside Macintosh: Processes,” Chapter 3, Time Manager.

Driver clients, such as compatibility shims, must also register with the USB Manager to receive notifications. A client registers with the USB Manager by calling the `USBInstallDeviceNotification` function, described in Chapter 6, “Callback Routine for Device Notification.”

The client notification messages are:

`kNotifyAddDevice` - USB device driver has been loaded

`kNotifyRemoveDevice` - USB device driver is about to be removed. After this message is processed, the `finalizeProc` for the device is called.

`kNotifyAddInterface` - USB interface driver has been loaded

`kRemoveInterface` - USB device driver that was loaded as an interface driver, is about to be removed. After this message is processed, the `finalizeProc` for the interface is called.

CHAPTER 4

Writing Mac OS USB Drivers

USB Services Library Reference

USB Services Library Reference

This chapter describes the APIs in the Mac OS USB Services Library that support software development of USB class drivers. This chapter does not provide any tutorial material designed to teach programming on the Macintosh platform. For information about how to program the Macintosh computer, see the documentation listed in “Supplemental Reference Documents” (page -15).

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 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.

The Mac OS USB DDK and other valuable resources for developers can be found at:

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

Note

The hub specific functions have been removed from this draft. Apple strongly recommends that developers of standard USB hub devices utilize the hub driver provided in the Mac OS USB software. The Mac OS hub driver supports hub devices that adhere to the version 1.0 and 1.1 USB specifications for standard hub devices. If you have a hub that you need to write a vendor specific device driver for, contact Apple Developer Technical Support at dts@apple.com and copy usb@apple.com.

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,

USB Services Library Reference

instantiate the appropriate pipe connections, determine device status, and perform read and write transactions with the device.

This section includes:

- a definition of the standard error reporting mechanism implemented by the USL
- a discussion of the concept of a USB reference used for the various device types found on the USB
- the USB parameter block used by the majority of USL functions
- guidelines for working in the asynchronous USL environment

Later sections define the functions provided in the USL, list the constants and common data structures, and finally provide a table of the errors returned through the USL.

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).

Error codes returned by the USL are in the range -6900 to -6999 and are listed in “USL Error Codes” (page 5-171).

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 errors. Transfer errors generally cause a pipe stall on non-default pipes. See text following the error list for exceptions. The transfer errors are in the range -6901 through -6915 as follows:

<code>kUSBCRCErr</code>	-6915	Bad CRC
<code>kUSBBitstuffErr</code>	-6914	Bitstuffing
<code>kUSBDataToggleErr</code>	-6913	Bad data toggle
<code>kUSBEndpointStallErr</code>	-6912	Device didn't understand
<code>kUSBNotRespondingErr</code>	-6911	No device, device hung
<code>kUSBPIDCheckErr</code>	-6910	PID CRC error

USB Services Library Reference

<code>kUSBWrongPIDErr</code>	-6909	Bad or wrong PID
<code>kUSBOverRunErr</code>	-6908	Packet too large or more data than allocated buffer
<code>kUSBUnderRunErr</code>	-6907	Less data than buffer, this error does not stall the pipe on isochronous transactions
<code>kUSBBufOvrRunErr</code>	-6904	Host hardware failure on data in, PCI busy?
<code>kUSBBufUnderRunErr</code>	-6903	Host hardware failure on data out, PCI busy?
<code>kUSBNotSent1Err</code>	-6902	Transaction not sent
<code>kUSBNotSent2Err</code>	-6901	Transaction not sent, this error does not stall the pipe on isochronous transfers

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.

The pipe is not stalled when a `kUSBNotSent2Err` is received if the number of any of the frames indicated in the frame list of an isochronous transfer have expired.

A `kUSBUnderRun` error does not stall the pipe on isochronous transactions. For example, the frame list that is constructed for audio input sets the request size to the maximum frame size that can be expected. Most frames return a request count that matches the unadjusted size with an occasional frame returning the adjusted size to make up for the fractional portion of the sample rate, for example, the point 1 in 44.1 kHz. All of the non-adjusted frames are less than the request size and result in an underrun error. This does not stall the pipe and requires no corrective action by the driver.

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:

<code>kUSBCRCErr</code>	-6915	Bad CRC
<code>kUSBBitsufErr</code>	-6914	Bitstuffing
<code>kUSBDataToggleErr</code>	-6913	Bad data toggle
<code>kUSBPIDCheckErr</code>	-6910	PID CRC error
<code>kUSBWrongPIDErr</code>	-6909	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 has been addressed in version 1.1 and later of the Mac OS USB software.

Under run errors occur when a packet shorter than the maximum packet size is received. It is the pipe policy to treat this situation as an error. Underrun errors are usually not generated. Currently, short packets always cause a normal completion.

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 6-177) 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 versions of the USBPB parameter block, the version 1.0 parameter block (`kUSBCurrentPBVersion`), and the version 1.1 parameter block (`kUSBIsocPBVersion`) that supports isochronous transfers. Version 1.1 and later of the Mac OS USB software accepts function calls made with both parameter blocks. For information about converting code that was dependent on the version 1.0 parameter block to use the 1.1 USBPB, see “Code Changes Required To Support The Version 1.1 USBPB” (page A-226).

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 USBPB parameter block 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;
```

USB Services Library Reference

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

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 usbHubBits {
    UInt32 Request;
    UInt32 Spare;
};

struct usbControlBits {
    UInt8 BMRquestType;      /* 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 = isochronous */
                           /* version 1.1 USBPB */
    UInt16 reserved1;      /* Reserved */
    UInt32 reserved2;      /* Reserved */
};
```

USB Services Library Reference

```

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

    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 */
    usbHubBits hub;             /* usbHubBits struct */
}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;

```

The older version 1.0 USBPB parameter block is defined as:

```

struct USBPB{

    void* qlink;
    UInt16 qType;
    UInt16 pbLength;             /* Length of parameter block */
}

```

USB Services Library Reference

```

    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;

```

USB Services Library Reference

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 `0x0100`. 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 or later (`kUSBIsoCPBVersion`).

The values passed in the `usbBMRequestType` field require a specific format, which can be derived by using the `USBMakeBMRequestType` function (page 5-120).

Required USB Parameter Block Fields

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

<code>pbLength</code>	Length of the <code>USBPB</code> parameter block, including any client additions
<code>pbVersion</code>	Version number of <code>USBPB</code> in binary-coded decimal, currently 1.1, initialize to <code>kUSBCurrentPBVersion</code> or <code>kUSBIsoCPBVersion</code> for isochronous call support. See “Changes In Mac OS USB Software” (page A-223) for additional information about how it use the isochronous variant of the USB parameter block.
<code>usbCompletion</code>	Completion routine
<code>usbRefcon</code>	For client use
<code>usbFlags</code>	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 5-1.

Table 5-1 Standard parameter block errors

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

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:

USB Services Library Reference

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

This structure encapsulates the transfer for one frame. On entry, the value of the `frReqCount` field is set to indicate how many bytes are to be transferred (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 transferred and the `frStatus` field specifies the result of the attempt.

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

<code>kUSBUnderRunErr</code>	-6907	Packet received was shorter than expected
<code>kUSBOverRunErr</code>		Packet too large or more data than buffer
<code>kUSBCRCErr</code>	-6908	
	-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 representative error if there are multiple packet errors.

The `usbBuffer` field points to the data, all of the packets to be sent or received are laid 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 `usbIsoBits` structure is a pointer to an array of `USBIsoFrame` structures that specify the individual packets. The start of any individual packet is found by adding the values in the

USB Services Library Reference

`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 5-113) is called. During a call to `USBConfigureInterface` function, the available bandwidth is checked. If bandwidth is insufficient (in versions of the USB software prior to 1.4), the call to open the isochronous pipes could fail. In version 1.4 and later, the pipe is created, but not in the active state. See the `USBConfigureInterface` function description for additional information.

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

USB Services Library Reference

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

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

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

--> <code>usbStatus</code>	Status information
--> <code>usbRefcon</code>	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 routine may use the same parameter block to make a new call to the USL. Polling the `usbStatus` field is not supported, since it may be used internally during function execution.

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.

USB Services Library Reference

The driver services library 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 work as described with USB version 1.1 and higher. For USB versions earlier than 1.1, the execution level can be checked with the `CurrentExecutionLevel` function, and the delay retried if the call back happens at a time other than task level. The `CurrentExecutionLevel` function is defined in the Driver Services Library chapter of *Designing PCI Cards and Drivers for Power Macintosh Computers*. The function returns a constant that defines the current execution level:

```
kHardwareInterruptLevel
kSecondaryInterruptLevel
kTaskLevel
```

An execution level of `kHardwareInterruptLevel` should not be seen for USL functions. If the current execution level is hardware interrupt level it indicates that something is not operating correctly.

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 level time (task level is described in *Designing PCI Cards and Drivers for the Power Macintosh Computer*) the system may still hang. In order to guard against this you should either:

- 1). Applications can give time back to the system by calling waitnext event. Shims should use SystemTask to give time back to the system.
- 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. 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.

Transaction and Data Timeouts

There are two types of timeout conditions on the USB, which can generally be referred to as “transaction” timeouts and “no data” timeouts.

- Transaction timeouts occur when too much time passes since a transaction became active.
- No data timeouts occur when too much time passes since the last data packet was transferred.

USB Services Library Reference

The clock for a transaction timeout starts when a transaction reaches the head of the endpoint queue. The transaction times out when the total amount of time the transaction is active is greater than the specified number of frames.

Transaction timeouts apply to the following functions:

```

USBDeviceRequest
USBControlRequest
USBBulkRead
USBBulkWrite
USBIntRead
USBIntWrite

```

Class drivers may request a transaction timeout by setting the `usbFlags` field to `kUSBTimeout` and specifying the time for the timeout in frames in the `usbFrame` field.

No data timeouts occur when there is no data flow on the pipe for the specified time. At the device level this means that the device has NAKed the request for the specified period of time. If any data transfer takes place (even a single ACKed transaction), the timeout clock is reset to the start. No data timeouts currently only apply to control transactions where the standard timeout is 5 seconds.

Class drivers may request a no data timeout by setting the `usbFlags` field to `kUSBNoDataTimeout` and specifying the time for the timeout in frames in the `usbFrame` field.

When a transaction has timed out, the pipe is aborted along with all pending transactions, and the `kUSBTimedout` error status is returned.

Starting with version 1.3 of the Mac OS USB software control transactions are subject to a 5 second no data timeout. This new default behavior can be overridden by a class driver by setting the `usbFlags` field in the USB parameter block to `kUSBNo5SecTimeout`. The use of `kUSBNo5SecTimeout` is not recommended. It should only be used in cases where the device is not compliant with the standard.

Driver specified timeouts are not meant to be totally accurate. Like other USB timing issues, the time value specified should be treated in a “no less than” manner. It may be longer than the specified value because of other system activity. No data timeouts are checked every 500ms, and specifying a value less than 500ms is of no use.

Setting both `kUSBTimeout` and `kUSBNoDataTimeout` is allowed, however the no data timeout cannot expire before the transaction timeout and therefore setting both is redundant. For control transactions, setting `usbFlags` to

`kUSBNoDataTimeout` and specifying a timeout value in `usbFrame` overrides the 5 second no data timeout without the need to set `kUSBNo5SecTimeout`.

Note

Isochronous transactions are time sensitive by nature and are not subject to the timeout mechanism described in this section. The timeout flags are not supported in isochronous transactions.

USL Functions

This section describes the functions in the USB Services Library.

Determining The Version of USB Software Present

Beginning with version 1.3 of the Mac OS USB software, the `USBGetVersion` function provides a way for drivers to determine the version number of Mac OS USB software present.

USBGetVersion

The `USBGetVersion` function allows device drivers to determine the version of Mac OS USB software that is running on the current Macintosh computer.

```
UInt32 USBGetVersion
        (void);
```

The `USBGetVersion` function 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.

USB Services Library Reference

mm	The next byte contains the minor version and revision number. The current value for the minor version number is 2. This number will increment with each minor release.
RR	The next byte contains the release stage. The release stage is defined as: 0x20 = development, 0x40 = alpha, 0x60 = beta, and 0x80 = final. If the software was at the beta release stage, this number would be 0x60.
ss	The least significant byte is the sequence number of the release, and it changes with every build of the USB software.

The following example demonstrates how this function may be used.

```
#include <USB.h> /* be sure that you are using the 1.3 USB.h file */

UInt32 MyUSBGetVersion (void)
{
    UInt32 version;
    if ((Ptr) USBGetVersion != (Ptr) kUnresolvedCFragSymbolAddress)
        version = USBGetVersion();
    else
        version = 0;          /* version of the USB less than 1.3 */
    return version;
}
```

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.

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 `USBSetConfiguration` function call to set the device configuration containing the interface. (The `USBSetConfiguration` function was called `USBOpenDevice` prior to version 1.1 of the Mac OS USB software)

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

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version number
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	Device reference
-- <code>usbBuffer</code>	Should be set to 0 (0 returned); reserved in this call
-- <code>usbActCount</code>	Should be set to 0 (0 returned); reserved in this call
--> <code>usbReqCount</code>	Maximum power requirement of the configuration, or 0 if not concerned about the power requirement. If the configuration requires more power than specified, the <code>kUSBDevicePowerProblem</code> error is returned. In versions of the Mac OS USB software prior to version 1.1, this field is not used and should be set to 0.
--> <code>usbFlags</code>	Should be set to 0 (0 returned); reserved in this call
<--> <code>usbClassType</code>	--> Class, 0 matches any class <-- Class value for interface found
<--> <code>usbSubclass</code>	--> Subclass, 0 matches any subclass <-- Subclass value for interface found

USB Services Library Reference

```

<--> usbProtocol    --> Protocol, 0 matches any protocol
                    <-- Protocol value for interface found

<--> usb.cntl.WValue
                    Configuration number; start with 0

<--> usb.cntl.WIndex
                    --> Interface number; start with 0
                    <-- Interface number

<--> usb0ther       Alternate interface, set to 0xff to find first alternate only

```

For alternate interface settings, the `usb0ther` field provides a method for getting details about a specific alternate interface or all of the alternate interfaces as a set. For example, the composite 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. If the appropriate amount of power is not available for the device, an error of `kUSBDevicePowerProblem` is returned. 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 `USBSetConfiguration` 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 wild cards 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.

USB Services Library Reference

The `usb.cntl.WValue` and `usb.cntl.WIndex` 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 `usb.cntl.WValue` and `usb.cntl.WIndex` 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

<code>paramErr</code>		<code>usbBuffer</code> pointer, <code>usbReqCount</code> , or <code>usbActCount</code> fields are not set to 0
<code>kUSBDevicePowerProblem</code>	-6976	an interface was found that matches the class, subclass, and protocol, but failed the power requirements. This error is also returned because no matching interfaces were found that meet the power requirement
<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBInternalErr,</code> <code>paramErr</code>	-6999	internal configuration descriptor cache corrupted
<code>kUSBNotFound</code>	-6987	interface or configuration specified is not in configuration descriptors

USBSetConfiguration, USBOpenDevice

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

For version 1.1 and greater of the Mac OS USB software `USBSetConfiguration` replaces the `USBOpenDevice` function name. The `USBSetConfiguration` and the `USBOpenDevice` functions have the same behavior, but `USBSetConfiguration` is the preferred function name, and should be used in versions of the Mac OS USB software 1.1 and greater.

The input and return values in the parameter block for the `USBSetConfiguration` are identical to those for the old `USBOpenDevice` function.

```
OSStatus USBOpenDevice(USBPB *pb); /* obsolete, do not use */
```

```
OSStatus USBSetConfiguration(USBPB *pb); /* for version 1.1 and later */
```

USB Services Library Reference

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version number
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	Device reference
-- <code>usbBuffer</code>	Should be set to 0 (0 returned); reserved in this call
-- <code>usbActCount</code>	Should be set to 0 (0 returned); reserved in this call
-- <code>usbReqCount</code>	Should be set to 0 (0 returned); reserved in this call
--> <code>usb.cntl.WValue</code>	Configuration number
--> <code>usbFlags</code>	Should be set to 0
<-- <code>usbOther</code>	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.

An interface reference cannot be used in the `usbReference` field to change the configuration. A device reference is required to implement a configuration change. However, an interface reference will not cause an error if used to specify the currently set configuration. This provides a level of backward compatibility.

Errors returned by the `USBOpenDevice` and `USBSetConfiguration` function include

<code>paramErr</code>		<code>usbBuffer</code> pointer, <code>usbReqCount</code> , or <code>usbActCount</code> fields are not set to 0
<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBDevicePowerProblem</code>	-6976	the device requires more power than is currently available. A power alert message will be displayed to the user. If the current driver does its own power error handling, it should not call <code>USBSetConfiguration</code> when the power is not available.

USB Services Library Reference

kUSBDeviceBusy	-6977	the device is already being configured
kUSBInternalErr,	-6999	internal configuration descriptor cache
paramErr		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 5-113) 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. An interface reference can be used in most circumstances where a device reference can be used. Individual function descriptions indicate where an interface or device reference cannot be used interchangeably.

```
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
--> usb.cntl.WIndex	Interface number

USB Services Library Reference

--> usbFlags Should be set to 0

If you create an interface reference, the interface reference must be disposed of in the driver finalize routine. In version 1.1 and later of the Mac OS USB software, the interface references are disposed of when a device is unplugged. However, you should still call the driver finalize routine and ignore any `kUSBUnknownDeviceErr` to ensure compatibility with all versions of the Mac OS USB software.

Errors returned by the `USBNewInterfaceRef` function include

<code>paramErr</code>		<code>usbBuffer</code> pointer, <code>usbReqCount</code> , or <code>usbActCount</code> fields are not set to 0
<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBDeviceBusy</code>	-6977	the device is already being configured
<code>kUSBNotFound</code>	-6987	interface or configuration specified is not in configuration descriptor
<code>kUSBOutOfMemoryErr</code>	-6988	ran out 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);
```

USB Services Library Reference

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

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

In version 1.4 and later of the USB software, when a pipe cannot be configured because of insufficient bandwidth available, it is still created, but not in an active state. In cases where bandwidth is insufficient, the `USBConfigureInterface` function returns a `kUSBNoBandwidthError` to indicate that one or more pipes failed because of insufficient bandwidth.

Errors returned by the `USBConfigureInterface` function include

<code>kUSBUnknownInterfaceErr</code>	-6978	<code>usbReference</code> does not refer to a current interface
<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>paramErr</code>		<code>usbBuffer</code> pointer, <code>usbReqCount</code> , or <code>usbActCount</code> fields are not set to 0
<code>kUSBInternalErr,</code> <code>paramErr</code>	-6999	internal configuration descriptor cache corrupted
<code>kUSBNotFound</code>	-6987	interface or configuration specified is not in configuration descriptor
<code>kUSBIncorrectTypeErr</code>	-6995	interface has control endpoints
<code>kUSBTooManyPipesErr</code>	-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 specifier 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
<-- usb.cntl.WValue
                  Maximum packet size of endpoint
```

This function takes either an interface or pipe reference in the `usbReference` field. To find the first pipe of a particular type, make a call to the function with

USB Services Library Reference

an interface reference. To find the next pipe of the same type, enter the pipe reference returned by the previous call.

Some additional explanation is in order with regard to the use of the phrase “the same type.” You may want to find a pipe with exactly the same type as that given in the previous call, either specific or wildcard, or find a pipe of exactly the same type as that returned by the last call, which is always a specific pipe type.

For example, you can search for any type and get a bulk-in pipe. After that, you could search for the next pipe of any type by using the returned pipe reference and the same wildcard values again, or you could search for the next bulk-in pipe by using the pipe reference returned from the last 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. Multiple iterations using a wildcard value require that the wildcard be set up for each subsequent call.

In version 1.4 and later of the USB software, if a pipe is created by the `USBConfigureInterface` function, but not active due to bandwidth constraints, the `USBFindNextPipe` function returns `kUSBNoBandwidthError`. The pipe reference returned in the `usbReference` field is valid, the pipe is not active though.

Errors returned by the `USBFindNextPipe` function include

<code>paramErr</code>		<code>usbBuffer</code> pointer, <code>usbReqCount</code> , or <code>usbActCount</code> fields are not set to 0
<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBNotFound</code>	-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
<--> usb.cntl.WIndex
                  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 `usb.cntl.WIndex` 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 `usb.cntl.WIndex` is passed back in the next call untouched, the function returns the next available matching descriptor.

USB Services Library Reference

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:

<code>kUSBUnknownInterfaceErr</code>	-6978	<code>usbReference</code> does not refer to a current interface or pipe
<code>kUSBInternalErr,</code> <code>paramErr</code>	-6999	internal configuration descriptor cache corrupted
<code>kUSBNotFound</code>	-6987	interface or configuration specified does not follow the starting index. A matching descriptor may already have been passed.

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

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

USB Services Library Reference

```

--> 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.

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

```

USB Services Library Reference

--> `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.

Note

The use of this function is optional if the `requestType` values are already provided in the standard documentation describing the device. The known `requestType` values can be transcribed directly into the `usbBMRequestType` field of the `USBDeviceRequest` function.

USB Services Library Reference

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.

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

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

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

<code>--> pbLength</code>	Length of parameter block
<code>--> pbVersion</code>	Parameter block version number
<code>--> usbCompletion</code>	The completion routine
<code>--> usbRefcon</code>	General-purpose value passed back to the completion routine
<code>--> usbReference</code>	The device reference passed to the driver when it is loaded
<code>--> usbBMRequestType</code>	The <code>usbBMRequestType</code> field is made up of the <code>direction</code> , <code>type</code> , and <code>recipient</code> values
<code>direction</code>	One of the following:
<code>kUSBIn</code>	Data will be transferred to the host.

USB Services Library Reference

	<code>kUSBOut</code>	Data will be transferred to the device.
	<code>kUSBNone</code>	No data will be transferred. The length and buffer parameters are ignored.
<code>type</code>		One of the following:
	<code>kUSBStandard</code>	A request defined in the USB standard.
	<code>kUSBClass</code>	A request defined in a class standard.
	<code>kUSBVendor</code>	A vendor unique request type.
<code>recipient</code>		One of the following:
	<code>kUSBDevice</code>	The request is to the whole device.
	<code>kUSBInterface</code>	The request is to a specific interface in the device.
	<code>kUSBEndpoint</code>	The request is to a specific pipe endpoint in a device.
	<code>kUSBOther</code>	The request is going somewhere else.
--> <code>usbRequest</code>		Defined by the USB standard, defined by a class driver standard, or vendor unique. See “ <code>usbRequest Constants</code> ” (page 5-165)
--> <code>usbWValue</code>		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.
--> <code>usbWIndex</code>		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.
--> <code>usbReqCount</code>		Specifies the size of the data to transfer. If this is set to 0, no transfer occurs
--> <code>usbBuffer</code>		Points to the data to be transferred (<code>kUSBOut</code> request) or where data will end up (<code>kUSBIn</code> request). The buffer should be at least as big as the size specified in the <code>usbReqCount</code> field. If <code>usbBuffer</code> is set to <code>nil</code> , no data is transferred in the data phase regardless of the value of length, setup and status still occurs
<-- <code>usbActCount</code>		Specifies the actual amount of data transferred on completion
--> <code>usbFlags</code>		Should be set to 0, or <code>kUSBAddressRequest</code> for control transactions addressed to an interface or endpoint. You may also use <code>kUSBTimeout</code> to indicate that you are setting the transaction timeout value according to the amount of frames in the <code>usbFrame</code> field. See

USB Services Library Reference

“Transaction and Data Timeouts” for additional information about transaction timeouts.

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, set in `usbFlags`, 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, specify the `kUSBAddressRequest` 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.

Drivers should not issue device requests for

```
Set_Address (bRequest = 5, type = standard, recipient = device)
Set_Config (bRequest = 9, type = standard, recipient = device)
```

These are reserved for system use. Issuing a `set_address` device request could disrupt communication on the USB. The `USBSetConfiguration` function should be used to set a configuration instead of a `set_config` device request through the `USBDeviceRequest` function. An attempt to issue a `set_config` with the `USBDeviceRequest` function will cause the `USBSetConfiguration` function to be called.

For additional information about the parameters specified for standard 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 `USBSetConfiguration` function. Other errors returned by the `USBDeviceRequest` function include

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBRqErr</code>	-6994	the value in the <code>usbBMRequestType</code> field is not valid
<code>kUSBUnknownRequestErr</code>	-6993	request code for a standard USB call is not recognized
<code>kUSBInvalidBuffer</code>		bad buffer specified

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 `USBPB` Parameter Block”.

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*.” It should be noted that memory returned by the USL `USBAllocMem` function is held as it is with `PoolAllocateResident`, and is therefore safe.

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 driver’s global space guarantees the memory will be held before requesting a transaction.

USBControlRequest

A general device control request can be issued on a non-default control pipe (a control pipe other than pipe 0) using the `USBControlRequest` function. The specified request is sent to the specified pipe’s endpoint and the relevant data is transferred. This function call can only be used after a device has been configured. See the `USBSetConfiguration` function description on (page 5-110), for additional information about configuring a device.

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

The fields required in the `USBPB` parameter block for the `USBControlRequest` function are:

```
--> pbLength      Length of parameter block
```

USB Services Library Reference

- > 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` call for the device you want to send the control request to. The device must have already been configured before a `USBControlRequest` is made.
- > usbBMRequestType
 - The `usbBMRequestType` field is made up of the `direction`, `type`, and `recipient` values (see also, the `USBMakeBMRequestType` function (page 5-120))
 - `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.
- > `usb.cntl.BRequest` Defined by the USB standard, or vender unique
- > `usb.cntl.WValue` General parameter unique to the transaction request

USB Services Library Reference

--> <code>usb.cntl.WIndex</code>	General parameter unique to the transaction request
--> <code>usbReqCount</code>	Specifies the size of the data to transfer. If this is set to zero, no data transfer will occur in the data phase.
--> <code>usbBuffer</code>	Points to the data to be transferred (<code>kUSBOut</code> request) or where data will end up (<code>kUSBIn</code> request). The buffer should be at least as big as the size specified in the <code>usbReqCount</code> field. If buffer is set to <code>nil</code> , no data is transferred in the data phase regardless of the value of length, setup and status still occur
<-- <code>usbActCount</code>	Specifies the actual amount of data transferred on completion
--> <code>usbFlags</code>	Should be set to 0, or <code>kUSBNo5SecTimeout</code> to disable the default 5-second timeout for control transactions. See “Transaction and Data Timeouts” for additional information about transaction timeouts.

In order to avoid the loss of data when transferring data from a device, `usbBuffer` and `usbReqCount` should be at least the `MaxPacket` size.

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

The errors returned by the `USBControlRequest` function are

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBRqErr</code>	-6994	the value in the <code>usbBMRequestType</code> field is not valid
<code>kUSBUnknownRequestErr</code>	-6993	request code for a standard USB call is not recognized
<code>kUSBInvalidBuffer</code>		bad buffer specified

USBIntRead

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

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

USB Services Library Reference

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

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

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

USBIntWrite

The `USBIntWrite` function queues an interrupt transaction on the specified pipe. The device is periodically polled and the transaction completes when the device has accepted the data in `usbActCount`. Interrupt out transactions were introduced in version 1.1 of the USB Specification. This function is only available in version 1.2 and later of the Mac OS USB software.

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

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

USB Services Library Reference

--> 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 <code>USBFindNextPipe</code> function,
--> usbReqCount	Specifies the size of the data to transfer, should be no longer than the <code>MaxPacketSize</code> of the endpoint
--> usbBuffer	Points to a buffer containing the outgoing data
<-- usbActCount	Specifies the actual amount of data transferred on completion
--> usbFlags	Should be set to 0

Errors returned by the `USBIntWrite` function include

<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBIncorrectTypeErr</code>	-6995	pipe is not an interrupt pipes
<code>kUSBPipeIdleErr</code>	-6980	specified pipe is in the idle state
<code>kUSBPipeStalledErr</code>	-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 <code>USBFindNextPipe</code> function

USB Services Library Reference

--> 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 may 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. See also "Bulk Data Transfer Performance Issues" (page A-230).

If you want less data than `MaxPacketSize`, it is still advisable to make a `MaxPacketSize` request, because the device is not aware of the actual amount of data requested, it just sees a request for more data. If `MaxPacketSize` is requested, any data transfer terminates the request. A short packet automatically terminates the request. If a request is made for `MaxPacketSize` and a `MaxPacketSize` size packet is returned, the transfer is also terminated, since the total amount of data requested is satisfied.

You should check the `usbActCount` field upon completion for the actual amount of data returned. If the value of `usbActCount` indicates that more or less data were returned than you expected, there may be a problem with your device. Requesting less than `MaxPacketSize` may cause a `kUSBOverRunErr` error to be returned if the device returns more data than was needed, in which case no data in the buffer is valid.

Errors returned by the `USBBulkRead` function include

<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBIncorrectTypeErr</code>	-6995	pipe reference is not a bulk-in pipe
<code>kUSBPipeIdleErr</code>	-6980	specified pipe is in the idle state
<code>kUSBPipeStalledErr</code>	-6979	specified pipe is stalled
<code>kUSBOverRunErr</code>	-6908	packet too large or more data than buffer

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version number
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	The pipe reference returned by the <code>USBFindNextPipe</code> or <code>USBSetConfiguration</code> functions
--> <code>usbReqCount</code>	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
--> <code>usbBuffer</code>	Points to the data to be transferred. The buffer should be at least as big as the size specified in the <code>usbReqCount</code> field. If the buffer is set to <code>nil</code> , no data is transferred regardless of the value of <code>usbReqCount</code>
<-- <code>usbActCount</code>	Specifies the actual amount of data transferred on completion
--> <code>usbFlags</code>	Should be set to 0

Errors returned by the `USBBulkWrite` function include

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

USBIsocRead

Isochronous data transfers are supported in version 1.2 and higher of the Mac OS USB software.

USB Services Library Reference

The `USBIsocRead` function supports isochronous read data transfers, and is defined as follows:

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

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version, must be <code>kUSBIsocPBVersion</code> for isochronous function calls
<-- <code>usbStatus</code>	Aggregate status, see discussion in “Using the USBPB For Isochronous Transactions”
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	The isochronous pipe reference returned by the <code>USBFindNextPipe</code> function
--> <code>usbReqCount</code>	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.
--> <code>usbBuffer</code>	Points to a buffer to which the incoming data is transferred
<-- <code>usbActCount</code>	Specifies the actual amount of data transferred on completion
--> <code>usbFlags</code>	Should be set to 0
--> <code>usb.isoc.FrameList</code>	Pointer to the list of frame structures
--> <code>frReqCount</code>	Number of bytes requested for each packet
<-- <code>frStatus</code>	Status returned by packet
<-- <code>frActCount</code>	Actual bytes transferred by packet
--> <code>usb.isoc.NumFrames</code>	Number of frames (specified in the <code>FrameList</code>) to attempt transfers in
--> <code>usbFrame</code>	Frame number of the first frame to transfer data

A complete discussion of how isochronous support is implemented and additional details about using the fields in the isochronous version of the

USBPB parameter block can be found in “Using the USBPB For Isochronous Transactions” (page 5-99).

USBIsocWrite

The `USBIsocWrite` function supports isochronous write transactions, and is defined as follows:

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

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version, must be <code>kUSBIsocPBVersion</code> for isochronous function calls
<-- <code>usbStatus</code>	Aggregate status, see discussion in “Using the USBPB For Isochronous Transactions”
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	The isochronous pipe reference returned by the <code>USBFindNextPipe</code> or <code>USBSetConfiguration</code> functions
--> <code>usbReqCount</code>	Specifies the size of the data to transfer. If this is set to 0 in the <code>frReqCount</code> field in the frame structure, the call sends a packet size of 0.
--> <code>usbBuffer</code>	Points to the data to be transferred. The buffer should be at least as big as the size specified in the <code>usbReqCount</code> field. If the buffer is set to <code>nil</code> , no data is transferred regardless of the value of <code>usbReqCount</code>
<-- <code>usbActCount</code>	Specifies the actual amount of data transferred on completion
--> <code>usbFlags</code>	Should be set to 0
--> <code>usb.isoc.FrameList</code>	Pointer to the list of frame structures
--> <code>frReqCount</code>	Number of bytes requested for each packet

USB Services Library Reference

```

        <-- 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

```

A complete discussion of how isochronous support is implemented and additional details about using the fields in the isochronous version of the USBPB parameter block can be found in “Using the USBPB For Isochronous Transactions”.

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

USB Services Library Reference

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.

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:

<code>kUSBActive</code>	Pipe can accept new transactions
<code>kUSBIdle</code>	Pipe cannot accept new transactions
<code>kUSBStalled</code>	An error occurred on the pipe

If the status is not active (`kUSBActive`), a non-zero status code (`kUSBPipeIdleErr` or `kUSBPipeStalledErr`) is also returned. If any other error is returned, the `state` variable is not changed.

Returning a non-zero status when the call succeeds, can make using this function a little tricky. This can be simplified if the pipe `state` variable is preloaded with a value not returned by the function, for example -1. The pipe `state` variable can then be examined to see if there was an error or to determine what state the pipe is in.

Errors returned by the `USBGetPipeStateByReference` function include:

<code>noErr</code>	0	specified pipe is active
<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBPipeIdleErr</code>	-6980	specified pipe is in the idle state
<code>kUSBPipeStalledErr</code>	-6979	specified pipe is stalled

In version 1.0 of the Mac OS USB 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 Mac OS USB software. If `noErr` is returned, the state is returned correctly. If 1.0 compatibility is desired, it can be worked around by examining the status returned. If this call returns an error, the error should be examined to see what state the pipe is in.

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 5-134) 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.

In version 1.2 and later of the Mac OS USB software, 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 5-134) 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-existent 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 `USBResetPipeByReference` 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 versions 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:

<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBPipeIdleErr</code>	-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:

<code>kUSBUnknownPipeErr</code>	-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 versions 1.0.1 and later of the USB Services software.

<code>kUSBPipeIdleStalled</code>	-6979	pipe was stalled, pipe is still active despite error
<code>kUSBPipeIdleErr</code>	-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.

USB Services Library Reference

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:

<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBPipeIdleStalled</code>	-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 versions 1.0.1 and later of the USB Services software.

<code>kUSBPipeIdleErr</code>	-6980	pipe was previously idle, pipe is still made active
------------------------------	-------	---

USBGetBandwidthAvailableByReference

The `USBGetBandwidthAvailableByReference` function, available in version 1.4 and later of the Mac OS USB software, determines the currently available bandwidth for isochronous and interrupt devices.

```
OSStatus USBGetBandwidthAvailableByReference(
    USBReference ref,
    UInt32 *avail)
```

--> `ref` Device, interface, or pipe reference

<-- `*avail` Currently available bandwidth in number of bytes per frame

The reference passed in specifies a bus to which the referenced object is attached. The bandwidth currently available for that bus is returned in `avail` as a number of bytes per frame.

The bandwidth value indicated is not guaranteed to be available when a request is made to reserve it. This is an indicator only.

Possible errors for the `USBGetBandwidthAvailableByReference` function include:

<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBPipeIdleErr</code>	-6980	specified pipe is in the idle state
<code>kUSBPipeStalledErr</code>	-6979	specified pipe is stalled

USBSetPipePolicy

The `USBSetPipePolicy` function, available in version 1.4 and later of the Mac OS USB software, provides a method for modifying pipe parameters.

```
OSStatus USBSetPipePolicy(USBPB *pb)
```

```
--> usbReference      Pipe reference
--> usbReqCount      Max data per IRP
--> usb.cntl.WValue  Max bytes per frame
--> usbOther        Max service interval
```

Field descriptions

`usbReqCount` The max data per IRP feature is not yet implemented, it may be used as a hint towards better resource management. If zero is passed it will not be changed. If it has never been specified, the default will be used.

IRP is equivalent to one call to a data transfer function, such as `USBBulkRead` or `USBIsocWrite`.

The default is probably 1MB, which is the maximum recommended to transfer in one call.

`usb.cntl.WValue` Max bytes per frame sets the maximum packet size that will be used on the pipe. This overrides that which is specified in the endpoint descriptor. The main use of this field is to allow isochronous endpoints to give back bandwidth they are not actually using. For isochronous pipes, valid values are 0 through 1023. A value of `0xffff` (`kUSBBandwidthUnchanged`) means not to change this from the current value.

If the bandwidth requested is not available, `kUSBNoBandwidthError` is returned.

USB Services Library Reference

This field cannot be used to increase the `MaxPacketSize` above that specified in the endpoint descriptor.

`usb0ther` Max Service interval is not yet implemented. This will be used with the isochronous stream model, when implemented. For Interrupt endpoints it can change the requested max polling interval. Valid values are 0-255. 0 meaning not to change it from the current setting. Low speed endpoints are not allowed to suggest a value in the range 1-7.

Errors returned by the `USBSetPipePolicy` function include:

<code>noErr</code>	0	specified pipe is active
<code>kUSBUnknownPipeErr</code>	-6997	pipe reference specified is unknown
<code>kUSBPipeIdleErr</code>	-6980	specified pipe is in the idle state
<code>kUSBPipeStalledErr</code>	-6979	specified pipe is stalled
<code>kUSBNoBandwidthError</code>		Indicates there was insufficient bandwidth to set the requested max packet size for the pipe.

Device Control and Status Functions

This section defines USL device control and status functions. The functions `USB_suspendDevice`, `USB_resumeDeviceByReference`, and `USB_portStatus` were introduced in version 1.4 of the Mac OS USB software, and as such, are only available in version 1.4 and later.

USBResetDevice

The `USBResetDevice` function resets a specified device. The port the device is attached to sends the reset signal for 10ms, as specified in the USB Specification. Following the reset signal, the device's USB address is set so that the USB device reference, `usbReference`, remains valid when the completion routine is called. The reset does not affect any other devices on the bus, unless the device reset is a hub. However, only hub drivers should be concerned with hubs.

This function should be considered a last resort to bring a misbehaving device back on line. If a device is reset, the driver will have to reinstate the device

USB Services Library Reference

configuration again, starting with getting a new interface reference. See “USBFindNextInterface” (page 5-108) for details.

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

Required fields in USBPB parameter block for the USBResetDevice 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	Set to 0

USBSuspendDevice

The USBSuspendDevice function sends a message to the hub driver requesting that the hub port the device is attached to be suspended.

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

Required fields in USBPB parameter block for the USBSuspendDevice 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 or interface reference
--> usbFlags	Set to 0

The USBSuspendDevice call completes asynchronously when the port is resumed. The port is resumed either by the device generating a USB RESUME signal, or by the driver issuing a USBResumeDeviceByReference call.

There are some hubs that cannot generate the correct SUSPEND/RESUME signaling for devices. These hubs immediately call the usbCompletion function indicating that they are resumed.

USBResumeDeviceByReference

The `USBResumeDeviceByReference` function causes a device which is suspended to be re-activated by generating the USB RESUME signal on the port the device is connected to.

```
OSStatus USBResumeDeviceByReference (usbReference ref)
```

ref Device or interface reference

This call completes immediately. However, the device is not activated and available until the completion routine from the `USBSuspendDevice` function is called. You can find out the current status of a port by using the `USBPortStatus` function.

USBPortStatus

This function queries the hub to determine the port status of the port that a particular device, interface, or pipe reference is connected to.

```
OSStatus USBPortStatus(USBPB *pb)
```

Required fields in `USBPB` parameter block for the `USBPortStatus` 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 pipe reference
--> usbFlags           Set to 0
<-- usbStatus          Port status
```

This call can return an immediate error of `kUSBDeviceBusy`, indicating that the hub driver cannot handle this request. Your driver should use the `USBDelay` function and try again.

Upon successful completion the `usbStatus` field contains one of the following values:

USB Services Library Reference

`kUSBDeviceDisconnected` - Device has been disconnected. Your driver can expect to be removed (since it hasn't been removed yet!)

`kUSBPortDisabled` - Port has been disabled. This is probably due to a babble condition. A babble condition is typically caused by electro-static discharge or other errors outside of your device). When the `kUSBPortDisabled` status is returned, you need to use `USBResetDevice` to recover the device.

`kUSBDeviceSuspended` - The port is currently suspended. It can be resumed using `USBResumeDeviceByReference` function.

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:

<code>kUSBBadDispatchTable</code>	-6950	improper driver dispatch table
<code>kUSBUnknownNotification</code>	-6949	notification type not defined
<code>kUSBQueueFull</code>	-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 *descUSBReference 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.

USB Services Library Reference

- > desc Device descriptor of the device to find a driver for. See also, “Device Descriptor Structure” (page 5-169)
- > 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 (`kUSB100mAavailable`) for a bus-powered hub parent and 500mA (`kUSB500mAavailable`) for a self-powered parent. See “USB Power and Bus Attribute Constants” (page 5-167).

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,
```

USB Services Library Reference

```
USBInterfaceDescriptor *interface,
USBReference hubRef,
UInt32 busPowerAvailable);
```

- > ref Device reference of device containing the interface.
- > desc Device descriptor of the interface to find a driver for. See also, "Device Descriptor Structure" (page 5-169)
- > interface Interface descriptor of interface to find a driver for. See also, "Interface Descriptor Structure" (page 5-170).
- > 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

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

Setting the `usbFlags` parameter to `kUSBTaskTimeFlag` requests that a call back be made at task level, and thus the callback only occurs at task level, and never at secondary interrupt time. This potentially means that the delay could be extended. Using the task time mechanism requires that the rest of the system software cooperates and gives the USB software time.

The `usbReqCount` field specifies the number of frames to delay. A requested delay of `kUSBNoDeLay` causes the call back to occur as soon as possible. When used in conjunction with the `kUSBTaskTimeFlag` flag, you can effect the quickest transition to task time.

It should be noted that the delay time requested is a minimum time. The actual delay time will never be less than the requested time, however it may be a longer delay depending upon other system activity. As noted in the previous discussion, using the `kUSBTaskTimeFlag` can cause the longest delays. Even if the `kUSBTaskTimeFlag` flag is not specified, the callback may be delayed if the USB Manager is not given time by the system.

USB Services Library Reference

The `USBDe1ay` function should not be used as a system-wide timing mechanism, since the time values are only relevant within the context of USB frames. The Mac OS USB software guarantees that only the specified number of frames will pass over the USB before the completion routine executes. It will never be less than the specified length of delay. Other activity affecting the system may determine how long it actually takes for a specified number of frames to pass. The functions in the Driver Services library provide accurate timing services for native drivers.

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 a call to the `USBDe1ay` function is unplugged and its driver removed while the function call is pending, the delay is cancelled and the function will not complete. In version 1.3 and later of the USB software, the delay will complete with a `kUSBAbortedError` if the `kUSBReturnOnException` flag is specified. Your finalize routine can assume that the delay is finished and safely dispose of the parameter block.

The `USBDe1ay` function returns the following errors:

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBAbortedError</code>	-6982	Pipe aborted. This error is returned when a delay call is pending and the associated device is unplugged

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);
```

USB Services Library Reference

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

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

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> 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.

USBAAllocMem

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

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

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

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version number
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	A device, interface, or pipe reference which associates the call with a device
--> <code>usbReqCount</code>	Amount of memory required to be allocated
<-- <code>usbActCount</code>	Amount of memory actually allocated
<-- <code>usbBuffer</code>	Memory allocated
--> <code>usbFlags</code>	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. In version 1.3 and later of the USB software, if the `kUSBReturnOnException` flag is specified, the delay completes with a `kUSBAbortedError`. Your `finalize` routine can assume that the delay is finished and safely dispose of the parameter block for this call.

USB Services Library Reference

The `USBAllocMem` function returns the following error:

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBAbortedError</code>	-6982	Pipe aborted. This error is returned when the call is pending and the associated device is unplugged

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:

--> <code>pbLength</code>	Length of parameter block
--> <code>pbVersion</code>	Parameter block version number
--> <code>usbCompletion</code>	The completion routine
--> <code>usbRefcon</code>	General-purpose value passed back to the completion routine
--> <code>usbReference</code>	A device, interface, or pipe reference which associates the call with a device
<--> <code>usbBuffer</code>	--> previously allocated memory to be deallocated <-- pointer set to nil
--> <code>usbFlags</code>	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, and the specified `usbReference` is not checked or used. 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 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. In version 1.3 and later of the USB software, if the `kUSBReturnOnException` flag is specified, the delay completes with a `kUSBAbortedError`. Your `finalize` routine can assume that the delay is finished and safely dispose of the parameter block for this call.

The `USBDeAllocMem` function returns the following error:

<code>kUSBUnknownDeviceErr</code>	-6998	<code>usbReference</code> does not refer to a current device
<code>kUSBCompletionError</code>	-6984	<code>kUSBNoCallback</code> was specified and current execution level is not task time
<code>kUSBAbortedError</code>	-6982	Pipe aborted. This error is returned when the call is pending and the associated device is unplugged

Byte Ordering (Endianism) Functions

There are several 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)
```

HostToUSBLong

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

```
UInt16 HostToUSBLong(UInt32 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.

USBToHostLong

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

```
UInt16 USBToHostLong(UInt32 value)
```

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 to the Expert status log. No weight is attached to this message by the operating system. The status messages can be seen by the USB Prober application in the USB Expert Log window.

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

--> `ref` Device reference for the device driver giving notification. The reference is for the purpose of displaying information only. Currently this reference is not validated.

--> `pointer` A pointer to a string to display. This value is a P-string.

--> `value` An arbitrary number to display.

USBExpertStatusLevel

The `USBExpertStatusLevel` function sends a message with a specific level of importance. The value for the level of status messages is sent to the USB Manager. These messages can be seen by the USB Prober application in the USB Expert Log window. The USB Prober also allows you to set the level of status messages you want displayed. Only messages below the current level are kept in the Expert log. For more information about the USB Expert Log window, see “USB Prober Features for Developers” (page 2-37).

```
OSStatus USBExpertStatusLevel (
    UInt32 level,
    USBDeviceRef ref,
    char *status,
    UInt32 value);
```

<code>level</code>	The level to assign to the status message. Integers 1 through 5 <code>kUSBStatusLevelFatal = 1</code> Fatal errors. <code>kUSBStatusLevelError = 2</code> General errors that may or may not effect operation. <code>kUSBStatusLevelClient = 3</code> General driver messages. <code>kUSBStatusLevelGeneral = 4</code> Important messages generated by the USB Expert and USL. <code>kUSBStatusLevelVerbose = 5</code> General messages from the USB Expert and USL.
<code>ref</code>	Device reference for the device driver giving notification. The reference is for the purpose of displaying information only. Currently this reference is not validated.
<code>status</code>	A pointer to a string status message to display. This value is a P-string.
<code>value</code>	An arbitrary number to display.

Note

The `USBExpertStatusLevel` function was added to the USL in Mac OS USB software version 1.2. Unless you weak link to this symbol, using this function prevents your driver from loading on systems running Mac OS USB software prior to version 1.2. ♦

USBExpertGetStatusLevel

The `USBExpertGetStatusLevel` function returns the current logging level. Only messages less than the returned status level are being recorded in the status log.

```
UInt32 USBExpertGetStatusLevel(void);
```

USBExpertSetStatusLevel

The `USBExpertSetStatusLevel` function set the current logging level. Only messages with a value lower than that specified are recorded in the status log.

```
void USBExpertSetStatusLevel(UInt32 level);
```

<code>level</code>	The status level values. Integers 1 through 5
<code>kUSBStatusLevelFatal = 1</code>	Fatal errors.
<code>kUSBStatusLevelError = 2</code>	General errors that may or may not effect operation.
<code>kUSBStatusLevelClient = 3</code>	General driver messages.
<code>kUSBStatusLevelGeneral = 4</code>	Important messages generated by the USB Expert and USL.
<code>kUSBStatusLevelVerbose = 5</code>	General messages from the USB Expert and USL.

The `USBExpertSetStatusLevel` function could be used to debug a call sequence at a higher level.

USBExpertFatalError

The `USBExpertFatalError` function is intended to inform the system of nonrecoverable errors in a class driver. Currently no action is taken when this

USB Services Library Reference

message is received other than to add the message to the expert log. 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. This value is a P-string.
- > 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 5-106).

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

USB Services Library Reference

`USBFindNextInterfaceDescriptorImmediate` and the `USBFindNextEndpointDescriptorImmediate` 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
--> usb.cntl.WValue
                  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 `usb.cntl.WValue` 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.

USB Services Library Reference

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
```

USB Services Library Reference

```

<--> usbSubclass    --> Subclass; 0 matches any subclass
                    <-- Subclass value for interface found

<--> usbProtocol    --> Protocol; 0 matches any protocol
                    <-- Protocol value for interface found

<--> usb.cntl.WValue
                    --> 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.

<-- usb.cntl.WIndex
                    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.

USB Services Library Reference

The errors returned by the `USBFindNextInterfaceDescriptorImmediate` function include:

<code>kUSBNotFound</code>	interface specified is not in configuration
<code>kUSBInternalErr</code> ,	not a valid configuration descriptor
<code>paramErr</code>	

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

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

USB Services Library Reference

```

<--> 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
<-- usb.cntl.WValue
                    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` 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:

<code>kUSBNotFound</code>	- 6987	endpoint specified is not in configuration
<code>kUSBInternalErr,</code> <code>paramErr</code>	- 6999	not a valid configuration descriptor

USBGetStringDescriptor

The `USBGetStringDescriptor` function is used to obtain the USB String Descriptor as follows

```
OSStatus USBGetStringDescriptor(USBPB *pb)
```

The relevant fields in the `USBPB` for the `USBGetStringDescriptor` function are:

```

--> usbReference      Device/Interface
--> usb.cntl.WValue   String number
--> usb.cntl.WIndex   Language code, 0 for English
--> usbBuffer         Where to put descriptor

```

USB Services Library Reference

```
--> usbReqCount
    Size of buffer
<-- usbActCount
    Size of returned string
```

Debugger Aware Flag

A debugger aware function `USLDebuggerActive` was introduced in version 1.4 of the Mac OS USB software. It is used primarily by system software to notify the USL that the system has entered or exited a debugger. However, the USL is aware of the flag, `kUSBDebuggerAwareFlag`, that can be set by USB drivers in transaction calls. This section describes the function, its behavior, and the use of the flag.

```
void USLDebuggerActive(Boolean active);
```

`active` When called with a true parameter the USL is notified that the debugger has been entered. All future transactions which complete, and are not marked as debugger aware, will be delayed until the debugger is exited.

When called with a false status, all transactions that had completed are dispatched back to the calling driver, and the USL does not delay transactions further.

USB drivers can set the `kUSBDebuggerAwareFlag` to make transactions immune to this behavior. If a driver wants to use the flag in a function call, it should check the current version of USB software before doing so. If this flag is set on a version of the Mac OS USB software prior to version 1.4, the transaction is rejected with a `kUSBFlagsError`. The `USBGetVersion` function is used to determine the current version of the USB software.

Deprecated Pipe Functions

The `USBOpenPipe` and `USBClosePipeByReference` functions have been deprecated from the current Mac OS USB API. Use of these functions is not supported for

future compatibility. Some of the old code samples in the Mac OS USB DDK used the `USBOpenPipe` and `USBClosePipeByReference` functions.

If you want your code to be compatible with future versions of the Mac OS USB software, use the functions defined in “USB Configuration Functions” (page 5-107) to configure a device interface and discover a pipe.

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*/
kUSBCurrentHubPB        = kUSBIsocPBVersion
```

Flag Constants

```
kUSBTaskTimeFlag        = 1,
kUSBHubPower            = 2,
kUSBPowerReset          = 4,
kUSBHubReaddress        = 8,
kUSBAddressRequest      = 16,
kUSBReturnOnException   = 32,
kUSBNo5SecTimeout       = 64,
kUSBTimeout             = 128,
kUSBNoDataTimeout       = 256,
kUSBDebugAwareFlag      = 512
```

USB Services Library Reference

Endpoint Type Constants

```

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

```

usbBMRequest Direction Constants

```

kUSBOut        = 0    /* Data is transferred to the device */
kUSBIn         = 1    /* Data is transferred to the host */
kUSBNone       = 2    /* No data is transferred */
kUSBAnyDirn    = 3    /* Any direction */

```

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

```

USB Services Library Reference

```

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
kUSBCommmClass        = 2
kUSBHIDClass          = 3
kUSBDisplayClass      = 4
kUSBPrintingClass     = 7
kUSBMassStorageClass  = 8
kUSBHubClass           = 9,
kUSBDataClass         = 10
kUSBVendorSpecificClass = 0xFF
};

```

USB Services Library Reference

Descriptor Type Constants

```

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

```

Feature Selector Constants

```

kUSBFeatureDeviceRemoteWakeup = 1
kUSBFeatureEndpointStall      = 0

```

Pipe State Constants

```

kUSBActive      = 0, /* Pipe can accept new transactions */
kUSBIdle        = 1, /* Pipe cannot accept new transactions */
kUSBStalled     = 2, /* An error occurred on the pipe */
kUSBSuspended  = 4, /* Device is suspended */
kUSBNoBandwidth = 8, /* Isochronous or Interrupt pipe could */
                       /* not be initialized due to bandwidth */
                       /* constraint */

```

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')

```

Driver Loading Option Constants

```

kUSBDoNotMatchGenericDevice = 0x00000001, /* Driver's VendorID */
                                        /* must match Device's */
                                        /* VendorID*/
kUSBDoNotMatchInterface = 0x00000002, /* Do not load this driver */
                                        /* as an interface driver.*/
kUSBProtocolMustMatch = 0x00000004, /* Do not load this driver */
                                        /* if protocol field */
                                        /* doesn't match.*/
kUSBInterfaceMatchOnly = 0x00000008 /* Only load this driver */
                                        /* as an interface driver.*/

```

Error Status Level Constant

```

kUSBStatusLevelFatal    = 1,
kUSBStatusLevelError    = 2,
kUSBStatusLevelClient   = 3,
kUSBStatusLevelGeneral  = 4,
kUSBStatusLevelVerbose  = 5

```

USB Data Structures

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

Device Descriptor Structure

The USB device descriptor is of this form:

```
struct USBDeviceDescriptor {
    UInt8      length;           /* Length of this descriptor */
    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;
};
```

Additional information about valid values for the fields in `USBDeviceDescriptor` structure can be found in the USB specifications.

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;
};
```

USB Services Library Reference

```

    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;
};

```

For more information about human interface devices (HIDs), see Chapter 7, “HID Library Reference.”

USL Error Codes

Error codes returned by the USL are in the range -6900 to -6999 as listed in Table 5-2.

Table 5-2 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

Table 5-2 Error definitions

Error constant	Number	Definition
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
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
kUSBInvalidBuffer	-6975	Bad buffer, usually nil
kUSBDeviceSuspended	-6974	Device is suspended
kUSBDeviceNotSuspended	-6973	Device is not suspended for resume

Table 5-2 Error definitions

Error constant	Number	Definition
kUSBDeviceDisconnected	-6972	Disconnected during suspend or reset
kUSBTimedOut	-6971	Transaction timed out. See “Transaction and Data Timeouts,” for additional details
kUSBBadDispatchTable	-6950	Improper driver dispatch table
kUSBUnknownNotification	-6949	Notification type not defined
kUSBQueueFull	-6948	Internal queue full
kUSBLinkErr	-6916	Link error
kUSBCRCErr	-6915	Pipe stall: bad CRC
kUSBBitstuffErr	-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
kUSBBufUnderRunErr	-6903	Host hardware failure on data out
kUSBNotSent1Err	-6902	Transaction not sent
kUSBNotSent2Err	-6901	Transaction not sent

USB Manager Reference

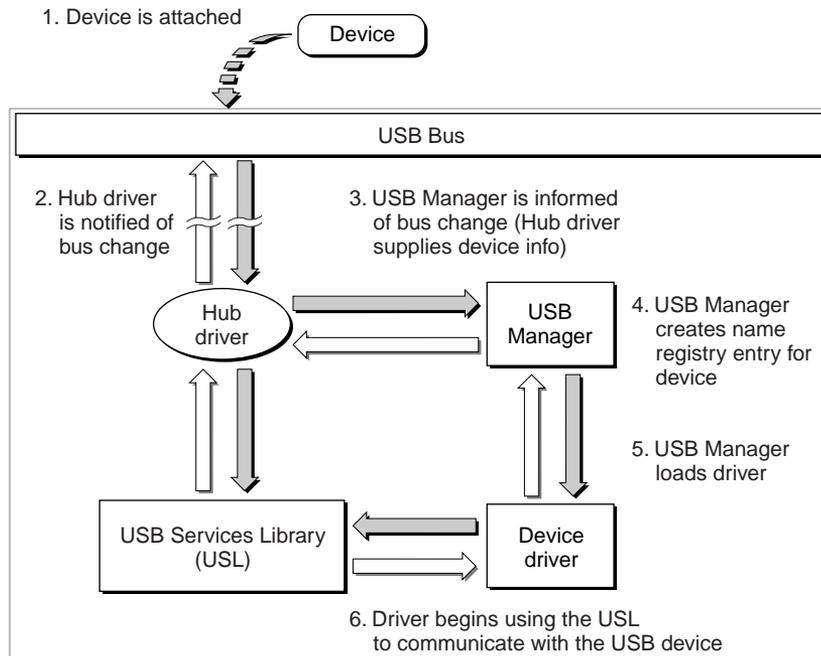
The USB Manager API is described in this chapter.

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 6-1 depicts the sequence of events that the USB Manager participates in when a device is added to the USB.

Figure 6-1 Device addition event sequence on the USB



USB Manager Reference

The USB Manager consists of native code fragments wrapped in a file of type 'expt'. A complete description of code fragments and the mechanisms for dealing with them in applications can be found in *Inside Macintosh: Power PC System Software, Chapter 3, "Code Fragment Manager."*

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 Mac OS ROM file, or in the case of machines without built-in USB ports, the USB Support extension 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 interface 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

USB Manager Reference

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 6-187).

Constant	Value	Description
<code>kUSBAnyClass</code>	<code>0xffff</code>	Pass in as a wildcard in the <code>deviceClass</code> parameter or <code>usbClass</code> field in the device notification parameter block.
<code>kUSBAnySubClass</code>	<code>0xffff</code>	Pass in as a wildcard in the <code>deviceSubClass</code> parameter or <code>usbSubClass</code> field in the device notification parameter block
<code>kUSBAnyProtocol</code>	<code>0xffff</code>	Pass in as a wildcard in the <code>deviceProtocol</code> parameter or <code>usbProtocol</code> field in the device notification parameter block
<code>kUSBAnyVendor</code>	<code>0xffff</code>	Pass in as a wildcard in the <code>usbVendor</code> field in the device notification parameter block.
<code>kUSBAnyClass</code>	<code>0xffff</code>	Pass in as a wildcard in the <code>usbProduct</code> 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 connection ID, `CFragConnectionID`, of the driver referenced by the device or interface reference.

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

--> `deviceRef` A pointer to the device or interface reference for which you want a connection ID.

<-- `connID` A pointer to the connection ID.

This function can be used to get the code fragment connection ID of a device driver or interface driver. An example of its use in an application would be to locate and display information about a device driver. You could use the `USBGetDriverConnectionID` function to get the connection ID, and then pass the connection ID to the `FindSymbol` function, defined in *Inside Macintosh: Power PC System Software, Chapter 3, "Code Fragment Manager,"* to locate the address of the `pTheUSBDriverDescription` structure, which contains the USB driver description information.

Here's a code snippet showing the basic idea. This snippet does not include any error or system zone checking code, which the driver would have to supply.

```
USBGetDriverConnectionID(&theDeviceRef, &connID);
FindSymbol (connID, "\pTheUSBDriverDescription",
    (Ptr *)&pTheUSBDriverDescription, &symClass);
sprintf((char *)buf, "Driver Description Version:. 0x%04x",
    pTheUSBDriverDescription->usbDriverDescVersion);
```

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.

USB Manager Reference

<-- busRef A pointer to the bus reference.

Passing Messages To Another Driver

The `USBDriverNotify` function supports message passing between drivers.

```
OSStatus USBDriverNotify (
    USBReference reference,
    USBDriverMessage msg,
    UInt32 refcon,
    USBDriverNotificationCallbackPtr callback);
```

--> reference A `USBReference` for the recipient driver.

--> msg The message to pass to the recipient driver

--> refcon General reference value passed the completion routine for use by the driver.

--> callback A pointer to a callback function.

Drivers call the `USBDriverNotify` function with the USB reference of the recipient driver, a message, a refcon, and an optional pointer to a callback function. The USB Manager locates the recipient driver based on the supplied reference and passes the message on by calling the `USBDeviceNotificationCallbackProc` for recipient driver, if it has a notification proc. Possible message constants are:

```
kNotifySystemSleepRequest = 0x00000001,
kNotifySystemSleepDemand = 0x00000002,
kNotifySystemSleepRevoke = 0x00000003
kNotifyHubEnumQuery = 0x00000006,
kNotifyChildMessage = 0x00000007,
kNotifyExpertTerminating = 0x00000008,
kNotifyDriverBeingRemoved = 0x0000000B
```

The sleep notification messages `kNotifySystemSleepRequest` and `kNotifySystemSleepDemand` are the same on any power-managed system. On PowerBook models, the processor and I/O subsystems are turned off when the machine goes to sleep. Because the USB is part of the I/O subsystem on PowerBooks with built-in USB or USB on a PC Card, the sleep state does effect USB drivers. Desktop Macintosh computers do not see these messages, since the processor and network I/O subsystems on desktop models remain active during system sleep.

USB Manager Reference

When a PowerBook computer with Mac OS USB software version 1.2 or later goes to sleep, the USB Manager sends any active USB drivers the `kNotifySystemSleepDemand` message and then unloads the driver just like a disconnect (hot unplug). When the PowerBook wakes up, the USB software re-enumerates the USB and appropriate drivers are loaded for any USB devices found on the bus.

Receiving A Message From A Child Driver

The `USBExpertNotifyParent` function allows a child driver to send a message to its parent driver.

```
USBExpertNotifyParent (
    USBReference reference,
    void * pointer);
```

--> `reference` The USB reference for the calling child driver.

--> `pointer` A pointer to a privately defined message for the parent driver.

If a parent driver has its `USBDeviceNotificationCallbackProc` called with a `kNotifyChildMessage`, then the parent driver should interpret the `pointer` argument as a privately defined message type from the child driver and the `refcon` argument as the `USBReference` of that child.

See the "Device Notification Parameter Block" (page 6-186) for details about information passed in the device notification callback.

Registering Shims After Boot Time

The `USBAddShimFromDisk` function allows extensions or installers to register a shim with the USB family expert after the system has booted. This function is available in version 1.3 and later of the Mac OS USB software.

```
USBAddShimFromDisk (
    FSSpec *shimFilePtr);
```

`shimFilePtr` The `FSSpec` file system specification record for the file containing the shim or shims. The Macintosh file system specification `FSSpec` record is defined in Chapter 2 of *Inside Macintosh: Files*.

The `USBAddShimFromDisk` function loads all code fragments that export the symbol "USBShim" from the file described by the `FSSpec`, and then calls the

USB Manager Reference

`USBShim` function. If the function returns a non-zero value, the code fragment is unloaded, otherwise it remains loaded while USB is active, typically until shutdown. However USB could be replaced, which would also cause shims to be unloaded. Code fragments and shims are discussed in Chapter 4, “Writing Mac OS USB Drivers.”

The `USBAddShimFromDisk` function is intended to be called by installers, or extensions that want to register a shim with the USB expert, but can’t do it at boot time because they require services that are not yet available. For example, they link to libraries such as `QuickTime` or `OpenTransport`.

This function provides a method for installers or extensions to load a shim in order to use a device immediately without the need to reboot. An example use for this function would be in a USB software updater/installer application. Such an installer could be designed to work when software is downloaded over the internet by installing the driver software, and using `USBAddShimFromDisk` function to activate any shims that were installed in the Extensions folder. The device could then begin to function immediately without rebooting the computer as is typically required for driver software with shims.

Note

This function is exported from the `USBFamilyExpertLib` and not from the `USBManagerLib`.

Adding a Driver For a Device After Boot Time

The `USBAddDriverForFSSpec` function adds a driver for a given device based on the device reference specified. Example uses for this function would be to load a specific driver for a device that doesn’t have a current driver loaded in the extensions folder, or to ensure that a vendor-specific driver is loaded for a vendor-specific device. This function is available in version 1.3 and later of the Mac OS USB software.

```
USBAddDriverForFSSpec (
    USBReference reference
    FSSpec *fileSpec);
```

`reference` **The `USBReference` for the device. Interface references are not permitted.**

`fileSpec` **Pointer to the `FSSpec` file system specification record for the driver file . The Macintosh file system specification `FSSpec` record is defined in Chapter 2 of *Inside Macintosh: Files*.**

USB Manager Reference

If there was a driver already installed for the reference, the routine returns the error `kUSBDeviceBusy` if the existing driver did not finalize immediately. When that happens, the USB software is deferring the termination of the driver. This means that it is the responsibility of the caller to give the system some task time so that Expert's idle task gets called and the deferred termination executed. If the caller bails out after the first call, the deferred termination will still happen (the old driver will eventually get unloaded).

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.

Note

Device notifications are only sent by the USB Manager when there are drivers currently loaded for the device or specific interface the notification request is registered for.

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

The callback routine is always called at task time, and may allocate memory, make Macintosh 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

--> pbLength	Length of parameter block
--> pbVersion	Version number of this parameter block
<--> usbDeviceNotification	The type of notification The following notifications are defined: kNotifyAnyEvent kNotifyAddDevice kNotifyAddInterface kNotifyRemoveDevice kNotifyRemoveInterface
--> reserved1[1]	Reserved, needed because of 2-byte 68k alignment
<-- 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

USB Manager Reference

<--> usbProtocol	The protocol of the target device, use <code>kUSBAnyProtocol</code> for any protocol
<--> usbVendor	The vendor ID of the target device, use <code>kUSBAnyVendor</code> for any vendor
<--> usbProduct	The product ID of the target device, use <code>kUSBAnyProduct</code> for any product
<-- result	The status of the call
<-- token	The value returned to uniquely identify this particular device notification. You pass this value to the <code>USBRemoveDeviceNotification</code> function.
--> 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 6-186).

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.

When registering with the USB Manager to be notified when device connections occur, you will also be notified when interfaces are connected to the USB stack. There is a differences between the notification implementation for devices connections and device driver loading and interface connections and interface driver loading. The difference is how the USB stack is informed about the connection.

With composite devices that are handled by the Apple composite driver, the USB stack is told about the interface by virtue of the load interface driver request. This occurs because the Apple composite driver discovers the interface

and asks the USB stack to handle it. For vendor specific drivers, the interfaces are not discovered by the Apple composite driver, and the USB stack is not informed or asked to handle the interface.

This means that if a shim or application installs a notification request into the USB Manager for an interface, and the device is handled by a vendor specific driver, then the shim or application won't be notified that the interface was discovered.

Unless a device is managed by the Apple composite driver, the shim or application will not be notified when a device with a particular interface is connected. Even if you have registered for notifications for that interface class and subclass, or if you are watching for a specific interface's class or subclass.

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.

Errors Returned By The USB Manager

Table 6-1 lists errors returned by the USB Manager.

Table 6-1 USB Manager error codes

<code>kUSBBadDispatchTable</code>	-6950	Improper driver dispatch table
<code>kUSBUnknownNotification</code>	-6949	Notification type not defined
<code>kUSBQueueFull</code>	-6948	Internal queue full

HID Library Reference

The Human Interface Device (HID) library APIs for the Mac OS are described in this chapter. To help understand the information presented in this chapter, you should be familiar with the material in the other chapters of this document, as well as the *Device Class Definitions for Human Interface Devices (HID)*, *HID Usage Table*, and *Usage Tables for HID Power Devices* class specifications, which are available at the USB organization web site. The USB class specifications can be downloaded at:

<http://www.usb.org/developers/index.html>

Overview

The Mac OS USB software includes a HID library that can determine the features of a HID device by parsing HID report descriptors. Examples of the types of HID devices the HID library can provide support for are:

- USB audio devices, such as external speakers with push button controls
- USB gaming devices, such as joysticks and gamepads
- Display monitors with USB ports and push button controls
- Uninterruptable power supplies with a USB host interface

The HID library is a shared library that is compatible with all Power Macintosh computers running Mac OS 8.1 or later. It does require that the Macintosh computer include a USB interface and have the Mac OS USB software version 1.2 or later installed. The USB hardware interface can be either built-in, like it is in the iMac computer, or in the form of a PCI card installed in the computer.

The HID library is primarily used by higher-level drivers or shims, such as an InputSprocket driver or the USB Mouse shim, to parse HID report descriptors and decode HID reports.

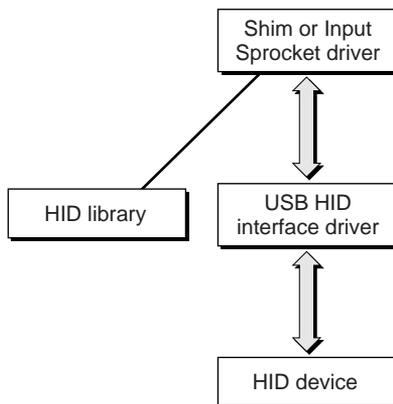
The block diagram in Figure 7-1 shows how HID library fits into the Mac OS USB software architecture. The diagram shows a HID device, the USB HID interface driver, the HID library, and a block which can be a high-level InputSprocket driver or USB shim. There are several additional Mac OS USB software components involved in making communication with the HID device possible. This diagram only shows those the higher-level driver generally deals with when using the HID library.

In this software model, the USB device is controlled by driver software. Apple system software provides drivers for standard devices, such as keyboards, mice, joysticks, and controls for audio, display, and uninterruptable power supplies.

For special features or devices, vendor specific drivers must be provided. If the device includes a Human Input Device, the USB HID interface driver exports a HID dispatch table for each device it supports and is matched against. The higher level software, such as a shim or InputSprocket driver, uses the device's HID interface driver API entry points defined in the dispatch table to communicate with the device. The API a driver needs to implement in the HID dispatch table is defined in "The HIDDeviceDispatchTable Structure" (page 4-76), and in the HID.h file.

The higher-level driver or shim software uses the HID library functions to obtain additional information about the features of the device. For example, an InputSprocket driver for a joystick will want to know what button features are available on the device and present that information to the user in a game environment. In order to find out what features are available, the InputSprocket driver will have to use the HID library functions to parse the HID report descriptor and decode the HID report for the joystick.

Figure 7-1 HID library in USB software architecture



In order for the higher level software to be informed of manual or programmatic changes to the human input device, it must use the driver's

HID Library Reference

`HIDDeviceDispatchTable` to install, and later remove, a report handler. The reports that then come to the handler are encoded by the driver according to the HID report descriptors. The HID Library functions are then used by the report handler to decode the reports. Conversely, when the higher level software needs to send information to the HID device, it must create a report and use the HID Library to encode the desired information. The HID driver does the actual work of sending the report to the device when the higher level software communicates with the driver through the `HIDDeviceDispatchTable`. Examples of how report handlers are implemented can be found in the HID Reader folder found in the Examples folder of the USB DDK.

HID Library API Reference

This section describes the functions supported by the HID Library. The functions are organized according to usage.

HID Descriptor Management Functions

The `HIDOpenReportDescriptor` and `HIDCloseReportDescriptor` functions are used for HID descriptor management.

`HIDOpenReportDescriptor`

```
extern OSStatus HIDOpenReportDescriptor (
    void * hidReportDescriptor,
    ByteCount descriptorLength,
    HIDPreparedDataRef * preparedDataRef,
    UInt32 flags);
```

The `HIDOpenReportDescriptor` function allocates the memory the parser needs to handle the given report descriptor, and then parses the report descriptor. A data reference to the parsed report information is returned.

HID Library Reference

`hidReportDescriptor`

Contains a pointer to the actual HID report descriptor from the USB device's firmware.

`descriptorLength`

The length of the HID report descriptor

`preparedDataRef`

Prepared data reference to be used for subsequent function calls

`flags`

Flags for this function are `kHIDFlag_StrictErrorChecking = 0x00000001`

When the parsed information is no longer needed, clients should call the `HIDCloseReportDescriptor` function.

HIDCloseReportDescriptor

```
extern OSStatus HIDCloseReportDescriptor (
    HIDPreparedDataRef preparedDataRef);
```

Disposes of the memory the parser allocated for the `HIDOpenReportDescriptor` function.

`preparedDataRef`

Prepared data reference for the report that is returned by the `HIDOpenReportDescriptor` function. After making a call to the `HIDCloseReportDescriptor` function, the `preparedDataRef` is invalid and should not be used.

HID Capabilities Functions

This sections defines the functions used to discover the capabilities of a HID device.

HIDGetButtonCaps

```
extern OSStatus HIDGetButtonCaps (
    HIDReportType reportType,
    HIDButtonCapsPtr buttonCaps,
    UInt32 * buttonCapsSize,
    HIDPreparedDataRef preparedDataRef);
```

Returns the button capabilities structures for a HID device based on the given prepared data.

reportType	Specifies the type or report for which to retrieve the scaled value. This parameter must be one of the following: kHIDInputReport, kHIDOutputReport, or kHIDFeatureReport
buttonCaps	Points to a caller-allocated buffer that will contain, on return, an array of HIDButtonCaps structures. The structures contain information for all buttons that meet the search criteria.
buttonCapsSize	
preparedDataRef	Prepared data reference for the report that is returned by the HIDOpenReportDescriptor function. After making a call to the HIDCloseReportDescriptor function, the preparedDataRef is invalid and should not be used.

HIDGetCaps

```
extern OSStatus HIDGetCaps (
    HIDPreparedDataRef preparedDataRef,
    HIDCapsPtr capabilities);
```

Returns a HIDCap capabilities structure in HIDCapsPtr. The HIDCap structure includes the number of capabilities of each type in a HID device. Based on the number for a given type, you can allocate the proper amount of memory to contain the structure for the specific capabilities you are interested in.

preparedDataRef

Prepared data reference for the HID device. This reference is returned by calling the `HIDOpenReportDescriptor` function.

capabilities

Points to a caller allocated buffer, that upon return contains the `HIDCap` capability structure for this HID device. The `HIDCap` structure is defined on (page 7-215).

HIDGetCollectionNodes

```
extern OSStatus HIDGetCollectionNodes (
    HIDCollectionNodePtr collectionNodes,
    UInt32 * collectionNodesLength,
    HIDPreparedDataRef preparedDataRef);
```

Returns an array of `HIDCollectionNode` structures that describe the relationships and layout of the link collections within this top level collection.

collectionNodes

Points to a caller-allocated array of `HIDCollectionNode` structures in which this routine returns an entry for each collection within the top level collection. A collection is a group of corresponding HID descriptors containing input, output, and feature items that have some common relationship to one another. For example, a pointer collection contains items for x and y position data, and button data. A HID collection is determined by the designer of the HID device

<--> collectionNodesLength

On input, specifies the length in array elements, of the buffer provided at `collectionNodes`. On output, this parameter is set to the number of entries in the `collectionNodes` array that were initialized.

preparedDataRef

The prepared data reference from `HIDOpenReportDescriptor`

The length of the buffer required, in array elements, for an entire collection node array is found in the `HIDCaps` structure member `numberCollectionNodes`. You obtain the `HIDCaps` information by calling the `HIDGetCaps` function.

For information on the relationships of link collections described by the data returned from this routine, see the description of the `HIDCollectionNode` structure.

HIDGetSpecificButtonCaps

```
extern OSStatus HIDGetSpecificButtonCaps (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    HIDButtonCapsPtr buttonCaps,
    UInt32 * buttonCapsLength,
    HIDPreparedDataRef preparedDataRef);
```

Retrieves the capabilities for all buttons in a specific type of report that meet the search criteria.

<code>reportType</code>	Specifies the type or report for which to retrieve the button capabilities. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>usagePage</code>	Specifies a usage page identifier to use as a search criteria. If this parameter is non-zero, then only buttons that specify this usage page will be retrieved.
<code>collection</code>	Specifies a link collection identifier to use as a search criteria. If this parameter is non-zero, then only buttons that are part of the specified link collection are retrieved.
<code>usage</code>	Specifies a usage identifier to use as a search criteria. If this parameter is non-zero, then only buttons that match the value specified are retrieved.
<code>buttonCaps</code>	Points to a caller-allocated buffer that will contain, on return, an array of <code>HIDButtonCaps</code> structures. The structures contain information for all buttons that meet the search criteria.
<code>buttonCapsLength</code>	On output, specifies the length, in array elements, of the buffer provided in the <code>buttonCaps</code> parameter. On output, this parameter is set to the actual number of elements that were

returned by the function call, in the buffer provided in the `buttonCaps` parameter, if the routine completed without error.

The correct length necessary to retrieve the button capabilities can be found in the capability data returned for the device by the `HIDGetCaps` function.

`preparedDataRef`

Prepared data reference returned by the `HIDOpenReportDescriptor` function

The `HIDGetSpecificButtonCaps` function retrieves capability data for buttons that meet a given search criteria, as opposed to the `HIDGetButtonCaps` function which returns the capability data for all buttons on the device. Calling this routine specifying zero for `usagePage`, `usage`, and `collection` is equivalent to calling the `HIDGetButtonCaps` function.

HIDGetSpecificValueCaps

```
extern OSStatus HIDGetSpecificValueCaps (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    HIDValueCapsPtr valueCaps,
    UInt32 * valueCapsLength,
    HIDPreparedDataRef preparedDataRef);
```

Retrieves the capabilities for all values in a specific type of report that meet the search criteria.

`reportType` Specifies the type or report for which to retrieve the value capabilities. This parameter must be one of the following: `kHIDInputReport`, `kHIDOutputReport`, or `kHIDFeatureReport`

`usagePage` Specifies a usage page identifier to use as a search criteria. If this parameter is non-zero, then only values that specify this usage page will be retrieved.

HID Library Reference

`collection` Specifies a link collection identifier to use as a search criteria. If this parameter is non-zero, then only values that are part of this link collection will be retrieved.

`usage` Specifies a usage identifier to use as a search criteria. If this parameter is non-zero, then only values that specify this usage will be retrieved.

`valueCaps` Points to a caller-allocated buffer that will contain, on return, an array of `HIDValueCaps` structures that contain information for all values that meet the search criteria

`valueCapsLength` Specifies the length on input, in array elements, of the buffer provided in the `valueCaps` parameter. On output, this parameter is set to the actual number of elements that were returned by this function call, in the buffer provided in `valueCaps` parameter, if the routine completed without error.

The correct length necessary to retrieve the value capabilities can be found in the capability data returned for the device from the `HIDGetCaps` function.

`preparedDataRef`
 Prepared data reference returned from the `HIDOpenReportDescriptor` function

The `HIDGetSpecificValueCaps` function retrieves capability data for values that meet given search criteria, as opposed to the `HIDGetValueCaps` function, which returns the capability data for all values on the device. Calling this routine with a value of zero for `usagePage`, `usage`, and `collection` parameters is equivalent to calling the `HIDGetValueCaps` function.

HIDGetValueCaps

```
extern OSStatus HIDGetValueCaps (
    HIDReportType reportType,
    HIDValueCapsPtr valueCaps,
    UInt32 * valueCapsSize,
    HIDPreparedDataRef preparedDataRef);
```

HID Library Reference

The `HIDGetValueCaps` function retrieves the capabilities for all values for a specified top level collection.

`reportType` Specifies the type or report for which to retrieve the value capabilities. This parameter must be one of the following: `kHIDInputReport`, `kHIDOutputReport`, or `kHIDFeatureReport`.

`valueCaps` On return, points to a caller-allocated buffer that contains an array of `HIDValueCaps` structures containing information for all values in the top level collection.

`valueCapsSize` On input, specifies the size in array elements of the buffer provided in the `valueCaps` parameter. On output, this parameter is set to the actual number of elements that were returned in the buffer provided in the `valueCaps` parameter, if the function completed without error.

The correct length necessary to retrieve the value capabilities can be found in the capability data returned for the device by the `HIDGetCaps` function.

`preparsedDataRef` The preparsed data reference returned from the `HIDOpenReportDescriptor` function.

The `HIDGetValueCaps` function retrieves the capability data for all values in a top level collection without regard for the usage, usage page, or collection of the value. To retrieve value capabilities for a specific usage, usage page, or collection, use the `HIDGetSpecificValueCaps` function.

HIDMaxUsageListLength

```
extern UInt32 HIDMaxUsageListLength (
    HIDReportType reportType,
    HIDUsage usagePage,
    HIDPreparsedDataRef preparsedDataRef);
```

The `HIDMaxUsageListLength` function returns the maximum number of buttons that can be returned from a given report type for the top level collection.

HID Library Reference

<code>reportType</code>	Specifies the type of report for which to get a maximum usage count. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>usagePage</code>	Optionally specifies the usage page identifier to use as a search criteria. If this parameter is zero, the function returns the number of buttons for the entire top-level collection regardless of the actual value of the usage page.
<code>preparedDataRef</code>	The prepared data reference for the report descriptor returned from the <code>HIDOpenReportDescriptor</code> function.

HID Report Decoding Functions

This section defines the functions used to decode HID reports. The functions described in this section do not get the values they return directly from the device. The values are extracted from the HID report. The typical way to get a value is to install a handler that passes along reports from the HID device at interrupt time. The reports are encoded. The functions in this section decode and extract the relevant data from the report.

HIDGetButtons

```
extern OSStatus HIDGetButtons (
    HIDReportType reportType,
    UInt32 collection,
    HIDUsageAndPagePtr * usageList,
    UInt32 * usageListSize,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDGetButtons` function takes a report from a HID device and returns the current state of the buttons in that report.

HID Library Reference

<code>reportType</code>	Specifies the type of report, provided in the <code>report</code> parameter, from which to retrieve the buttons. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>collection</code>	Optionally specifies the link collection identifier used to retrieve only specific button states. If this value is non-zero, only the buttons that are part of the given collection are returned.
<code>usageList</code>	On return, points to a caller-allocated buffer that contains the usages of all the buttons that are pressed.
<code>usageListSize</code>	Is the size, in array elements, of the buffer provided in the <code>usageList</code> parameter. On return, this parameter contains the number of button states that were set by this routine. If the error <code>kHIDBufferTooSmallErr</code> was returned, this parameter contains the number of array elements required to hold all button data requested. The maximum number of buttons that can ever be returned for a given type of report can be obtained by calling the <code>HIDMaxUsageListLength</code> function.
<code>preparedDataRef</code>	Prepared data reference returned from the <code>HIDOpenReportDescriptor</code> function
<code>report</code>	Points to the caller-allocated buffer that contains the device report data
<code>reportLength</code>	Specifies the length, in bytes, of the report data provided in the <code>report</code> parameter.

If you want to get only the buttons that appear on a particular page, you use the `HIDGetButtonsOnPage` function.

HIDGetButtonsOnPage

```
HIDGetButtonsOnPage(
    HIDReportType reportType,
    HIDUsage usagePage,
```

HID Library Reference

```

    UInt32 collection,
    HIDUsage * usageList,
    UInt32 * usageListSize,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);

```

Retrieves the button state information for buttons on a specified usage page.

<code>reportType</code>	Specifies the type of report, provided in the <code>report</code> parameter, from which to retrieve the buttons. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>usagePage</code>	Specifies the usage page of the buttons for which to retrieve the current state.
<code>collection</code>	Optionally specifies the link collection identifier used to retrieve only specific button states. If this value is non-zero, only the buttons that are part of the given collection are returned.
<code>usageList</code>	On return, points to a caller-allocated buffer that contains the usages of all the buttons that are pressed and belong to the usage page specified in the <code>usagePage</code> parameter.
<code>usageListSize</code>	Is the size, in array elements, of the buffer provided in the <code>usageList</code> parameter. On return, this parameter contains the number of button states that were set by this routine. If the error <code>kHIDBufferTooSmallErr</code> was returned, this parameter contains the number of array elements required to hold all button data requested. The maximum number of buttons that can ever be returned for a given type of report can be obtained by calling the <code>HIDMaxUsageListLength</code> function.
<code>preparedDataRef</code>	Prepared data reference returned from the <code>HIDOpenReportDescriptor</code> function
<code>report</code>	Points to the caller-allocated buffer that contains the device report data

HID Library Reference

`reportLength` Specifies the size, in bytes, of the report data provided in the `report` parameter.

HIDGetScaledUsageValue

```
extern OSStatus HIDGetScaledUsageValue (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    SInt32 * usageValue,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDGetScaledUsageValue` function returns the capabilities for all buttons for a given top level collection.

<code>reportType</code>	Specifies the type or report for which to retrieve the scaled value. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>usagePage</code>	Specifies the usage page of the value to be retrieved.
<code>collection</code>	Optionally specifies the link collection identifier of the value to be retrieved.
<code>usage</code>	Specifies the usage of the scaled value to be retrieve.
<code>usageValue</code>	Points to a variable, that on return from this routine holds the scaled value retrieved from the device report.
<code>preparedDataRef</code>	Prepared data reference from <code>HIDOpenReportDescriptor</code>
<code>report</code>	Points to the caller-allocated buffer that contains the device report data
<code>reportLength</code>	Specifies the length, in bytes, of the report data provided at report.

HID Library Reference

Clients who wish to obtain all capabilities for a `usage` that contains multiple data items for a single `usage` that corresponds to a HID byte array, must call the `HIDGetUsageValueArray` function.

HIDGetUsageValue

```
extern OSStatus HIDGetUsageValue (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    SInt32 * usageValue,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDGetUsageValue` function returns a value from a device data report given a selected search criteria.

<code>reportType</code>	Specifies the type of report, provided at <code>report</code> , from which to retrieve the value. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code>
<code>usagePage</code>	Specifies the usage page of the value to retrieve.
<code>collection</code>	Optionally specifies the link collection identifier of the value to be retrieved.
<code>usage</code>	Specifies the usage of the value to be retrieve.
<code>usageValue</code>	Points to a variable, that on return from this routine holds the value retrieved from the device report.
<code>preparedDataRef</code>	The prepared data reference obtained by calling the <code>HIDOpenReportDescriptor</code> function.
<code>report</code>	Points to the caller-allocated buffer that contains the device report data.
<code>reportLength</code>	Specifies the size, in bytes, of the report data provided in the <code>report</code> parameter.

HID Library Reference

The `HIDGetValue` function does not sign the value. To have the sign bit automatically applied, use the `HIDGetScaledUsageValue` function instead. For manually assigning the sign bit, the position of the sign bit can be found in the `HIDValueCaps` structure for this value.

Clients who wish to obtain all data for a usage that contains multiple data items for a single usage, corresponding to a HID byte array, must call the `HIDGetValueArray` function instead.

HIDGetValueArray

```
extern OSStatus HIDGetValueArray (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    Byte * usageValueBuffer,
    ByteCount usageValueBufferSize,
    HIDPreparsedDataRef preparsedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDGetValueArray` function returns a value from a device data report given a selected search criteria.

<code>reportType</code>	Specifies the type of report, provided at report, from which to retrieve the values. This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code> .
<code>usagePage</code>	Specifies the usage page of the data to be retrieved.
<code>collection</code>	Optionally specifies the link collection identifier of the data to be retrieved.
<code>usage</code>	Specifies the usage identifier of the value to be retrieve.
<code>usageValueBuffer</code>	Points to a caller-allocated buffer that contains, on output, the data from the device. The correct length for this buffer can be found by multiplying the <code>reportCount</code> and <code>bitSize</code> fields of the <code>HIDValueCaps</code> structure for this value and rounding the resulting value up to the nearest byte.

HID Library Reference

- `usageValueBufferSize` Specifies the size, in bytes, of the buffer in the `usageValueBuffer` parameter.
- `preparedDataRef` The prepared data reference returned from the `HIDOpenReportDescriptor` function.
- `report` Points to the caller-allocated buffer that contains the device report data.
- `reportLength` Specifies the size, in bytes, of the report data provided at report.

When the `HIDGetUsageValueArray` function retrieves the data, it fills in the buffer in little-endian order beginning with the least significant bit of the data for this usage. The data is filled in without regard to byte alignment and is shifted such that the least significant bit is placed as the 1st bit of the given buffer.

HIDUsageListDifference

```
extern OSStatus HIDUsageListDifference (
    HIDUsage * previousUsageList,
    HIDUsage * currentUsageList,
    HIDUsage * breakUsageList,
    HIDUsage * makeUsageList,
    HIDUsage usageListsLength);
```

The `HIDUsageListDifference` function compares and provides the differences between two lists of buttons.

- `previousUsageList` Points to the older button list to be used for comparison.
- `currentUsageList` Points to the newer button list to be used for comparison.
- `breakUsageList` On return, points to a caller allocated buffer that contains the buttons set in the older list, specified in the `previousUsageList` parameter, but not set in the new list, specified in the `currentUsageList` parameter.

HID Library Reference

`makeUsageList`

On return, points to a caller allocated buffer that contains the buttons set in the new list, specified in the `currentUsageList` parameter, but not set in the old list, specified in the `previousUsageList` parameter.

`usageListsLength`

Specifies the length, in array elements, of the buffers provided in the `currentUsageList` and `previousUsageList` parameters.

HID Report Data Encoding Functions

The functions defined in this section are used to encode data to be sent back to the device through HID reports.

HIDSetButton

```
extern OSStatus HIDSetButton (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    HIDPreparsedDataRef preparsedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDSetButton` function takes a report from a HID device and sets the current state of the specified button in that report.

`reportType` Specifies the type of report, provided at report. This parameter must be one of the following: `kHIDInputReport`, `kHIDOutputReport`, or `kHIDFeatureReport`.

`usagePage` Specifies the usage page identifier of the value to be set in the report.

`collection` Optionally specifies the link collection identifier to distinguish between buttons. If this parameter is zero, it is ignored.

HID Library Reference

usage	Points to a caller-allocated buffer that contains the button data to be set in the report in the <code>report</code> parameter.
preparedDataRef	The prepared data reference for the report descriptor returned by the <code>HIDOpenReportDescriptor</code> function.
report	Points to the caller-allocated buffer that contains the device report data.
reportLength	Specifies the size, in bytes, of the report data provided in the <code>report</code> parameter.

HIDSetButtons

```
extern OSStatus HIDSetButtons (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage * usageList,
    UInt32 * usageListSize,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDSetButtons` function takes a report from a HID device and returns the current state of the buttons in that report.

reportType	Specifies the type of report, provided at <code>report</code> . This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code> .
usagePage	Specifies the usage page identifier of the value to be set in the report.
collection	Optionally specifies the link collection identifier to distinguish between buttons. If this parameter is zero, it is ignored.
usageList	Points to a caller-allocated buffer that contains an array of button data to be set in the report in the <code>report</code> parameter.

HID Library Reference

usageListSize

Specifies the size, in array elements, of the buffer provided in the `usageList` parameter. If an error is returned by a call to this function, the `usageListLength` parameter contains the location in the array provided in the `usageList` parameter where the error was encountered. All array entries encountered prior to the error location were successfully set in the report provided in the `report` parameter.

preparsedDataRef

The preparsed data reference for the report descriptor returned by the `HIDOpenReportDescriptor` function.

report

Points to the caller-allocated buffer that contains the device report data.

reportLength

Specifies the size, in bytes, of the report data provided in the `report` parameter.

HIDSetScaledUsageValue

```
extern OSStatus HIDSetScaledUsageValue (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    SInt32 usageValue,
    HIDPreparsedDataRef preparsedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDSetScaledUsageValue` function takes a signed physical (scaled) number and converts it to the logical, or device representation and inserts it in a given report.

reportType

Specifies the type of report, provided at report. This parameter must be one of the following: `kHIDInputReport`, `kHIDOutputReport`, or `kHIDFeatureReport`

usagePage

Specifies the usage page identifier of the value to be set in the report.

HID Library Reference

<code>collection</code>	Optionally specifies the link collection identifier to distinguish between values that have the same usage page and usage identifiers. If this parameter is zero, it will be ignored.
<code>usage</code>	Specifies the usage identifier of the value to be set in the report.
<code>usageValue</code>	Specifies the physical, or scaled, value to be set in the value for the given report.
<code>preparedDataRef</code>	The prepared data reference for the report descriptor returned from the <code>HIDOpenReportDescriptor</code> function.
<code>report</code>	Points to the caller-allocated buffer that contains the device report data.
<code>reportLength</code>	Specifies the length, in bytes, of the report data specified in the <code>report</code> parameter.

The `HIDSetScaledUsageValue` function automatically handles the setting of the signed bit in the data to be sent to the device.

HIDSetUsageValue

```
extern OSStatus HIDSetUsageValue (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    SInt32 usageValue,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDSetUsageValue` function sets a value in a given report.

<code>reportType</code>	Specifies the type of report, provided at <code>report</code> . This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code> .
<code>usagePage</code>	Specifies the usage page identifier of the value to be set in the report.

HID Library Reference

<code>collection</code>	Optionally specifies the link collection identifier to distinguish between values that have the same usage page and usage identifiers. If this parameter is zero, it is ignored.
<code>usage</code>	Specifies the usage identifier of the value to be set in the report.
<code>usageValue</code>	Specifies the data this is to be set in the value for the given report.
<code>preparedDataRef</code>	The prepared data reference for the report descriptor returned from the <code>HIDOpenReportDescriptor</code> function.
<code>report</code>	Points to the caller-allocated buffer that contains the device report data.
<code>reportLength</code>	Specifies the size, in bytes, of the report data provided in the <code>report</code> parameter.

The `HIDSetUsageValue` function does not automatically handle the sign bit. Clients must either manually set the sign bit, at the position provided in the `HIDValueCaps` structure for this value, or call the `HIDSetScaledUsageValue` function.

HIDSetUsageValueArray

```
extern OSStatus HIDSetUsageValueArray (
    HIDReportType reportType,
    HIDUsage usagePage,
    UInt32 collection,
    HIDUsage usage,
    Byte * usageValueBuffer,
    ByteCount usageValueBufferLength,
    HIDPreparedDataRef preparedDataRef,
    void * report,
    ByteCount reportLength);
```

The `HIDSetUsageValue` function sets an array of values in a given report.

<code>reportType</code>	Specifies the type of report, provided at <code>report</code> . This parameter must be one of the following: <code>kHIDInputReport</code> , <code>kHIDOutputReport</code> , or <code>kHIDFeatureReport</code> .
-------------------------	---

HID Library Reference

<code>usagePage</code>	Specifies the usage page identifier of the value to be set in the report.
<code>collection</code>	Optionally specifies the link collection identifier to distinguish between values that have the same usage page and usage identifiers. If this parameter is zero, it is ignored.
<code>usage</code>	Specifies the usage identifier of the value to be set in the report.
<code>usageValueBuffer</code>	Points to a caller-allocated buffer that contains, on output, the data from the device. The correct length for this buffer can be found by multiplying the <code>reportCount</code> and <code>bitSize</code> fields of the <code>HIDValueCaps</code> structure for this value and rounding the resulting value up to the nearest byte.
<code>usageValueBufferLength</code>	Specifies the size, in bytes, of the buffer in the <code>usageValueBuffer</code> parameter.
<code>preparedDataRef</code>	The prepared data reference for the report descriptor returned from the <code>HIDOpenReportDescriptor</code> function.
<code>report</code>	Points to the caller-allocated buffer that contains the device report data.
<code>reportLength</code>	Specifies the size, in bytes, of the report data provided in the <code>report</code> parameter.

The `HIDSetUsageValue` function does not automatically handle the sign bit. Clients must either manually set the sign bit, at the position provided in the `HIDValueCaps` structure for this value, or call the `HIDSetScaledUsageValue` function.

HID Library Constants

This section lists the constants associated in the HID library APIs.

HID Report Constants

```

kHIDInputReport      = 1,
kHIDOutputReport     = 2,
kHIDFeatureReport    = 3,
kHIDUnknownReport    = 255

```

HIDOpenReportDescriptor Flags

```

kHIDFlag_StrictErrorChecking = 0x00000001

```

HIDGetDeviceInfo Constants

The constants listed below are passed to the `HIDGetDeviceInfo` function, which is implemented in the `TheHIDModuleDispatchTable`.

```

kHIDGetInfo_VendorID           = 1,
kHIDGetInfo_ProductID         = 2,
kHIDGetInfo_VersionNumber      = 3,
kHIDGetInfo_MaxReportSize     = 0x10,
kHIDGetInfo_GetManufacturerString = 0x0100,
kHIDGetInfo_GetProductString   = 0x0101,
kHIDGetInfo_GetSerialNumberString = 0x0102,
kHIDGetInfo_GetIndexedString   = 0x0103,
kHIDGetInfo_VendorSpecificStart = 0x00010000

```

HIDOpenDevice Constants

These permission constants are passed to the `HIDControlDevice` function.

```

kHIDPerm_ReadOnly           = 0x0001,
kHIDPerm_ReadWriteShared    = 0x0003,
kHIDPerm_ReadWriteExclusive = 0x0013

```

HIDInstallReportHandler Constant

```
kHIDFlag_CallbackIsResident = 0x0001
```

HIDControlDevice Constant

```
kHIDVendorSpecificControlStart = 0x00010000
```

Usage Table Constants

Usage table constants can be found in the *HID Usage Table* class specification and in the `HID.h` file.

HID Library Data Structures

This section lists the data structures that are associated with the HID Library.

HIDUsageAndPage

```
struct HIDUsageAndPage
{
    HIDUsage      usage;
    HIDUsage      usagePage;
};

typedef HIDUsageAndPage * HIDUsageAndPagePtr;
```

The `HIDUsageAndPage` data structure is used by HID clients when obtaining status of buttons to hold the usage page and usage of a button that is down.

Field descriptions

<code>usage</code>	Specifies the usage identifier within the usage page specified by <code>usagePage</code> of a button that is down.
--------------------	--

HID Library Reference

`usagePage` Specifies the usage page identifier of a button that is down. Clients use the `HIDUsageAndPage` structure with the `HIDGetButtonsEx` function to obtain both the usage page and usage identifiers of each button that is down.

HIDCaps

```
struct HIDCaps
{
    HIDUsage    usage;
    HIDUsage    usagePage;
    UInt32      inputReportByteLength;
    UInt32      outputReportByteLength;
    UInt32      featureReportByteLength;
    UInt32      numberCollectionNodes;
    UInt32      numberInputButtonCaps;
    UInt32      numberInputValueCaps;
    UInt32      numberOutputButtonCaps;
    UInt32      numberOutputValueCaps;
    UInt32      numberFeatureButtonCaps;
    UInt32      numberFeatureValueCaps;
};
```

```
typedef HIDCaps * HIDCapsPtr;
```

The `HIDCaps` data structure is used by HID clients to hold the capabilities of a HID device.

Field descriptions

`usage` Specifies the specific class of functionality that this device provides. This value is dependent and specific to the value provided in the `usagePage` field. For example, a keyboard could have a `usagePage` of `kHIDUsagePage_Generic` and a `usage` of `kHIDUsage_Generic_Keyboard`.

`usagePage` Specifies the usage page identifier for this top level collection.

`inputReportByteLength` Specifies the maximum length, in bytes, of an input report

HID Library Reference

for this device, including the report ID which is unilaterally prepended to the device data.

`outputReportByteLength`

Specifies the maximum length, in bytes, of an output report for this device, including the report ID which is unilaterally prepended to the device data.

`featureReportByteLength`

Specifies the maximum length, in bytes, of a feature report for this device, including the report ID which is unilaterally prepended to the device data.

`numberOfCollectionNodes`

Specifies the number of `HIDCollectionNode` structures that are returned for this top level collection by the `HIDGetConnectionNodes` function.

`numberOfInputButtons`

Specifies the number of input buttons.

`numberOfInputValues`

Specifies the number of input values.

`numberOfOutputButtons`

Specifies the number of output buttons.

`numberOfOutputValues`

Specifies the number of output values.

`numberOfFeatureButtons`

Specifies the number of feature buttons.

`numberOfFeatureValues`

Specifies the number of feature values.

This structure holds the parsed capabilities and data maximums returned for a device by the `HIDGetCaps` function.

HIDCollectionNode

```
struct HIDCollectionNode
{
    HIDUsage    collectionUsage;
    HIDUsage    collectionUsagePage;
    UInt32      parent;
```

HID Library Reference

```

    UInt32    numberOfChildren;
    UInt32    nextSibling;
    UInt32    firstChild;
};
typedef HIDCollectionNode * HIDCollectionNodePtr;

```

HIDButtonCaps

```

struct HIDButtonCaps
{
    HIDUsage    usagePage;
    UInt32     reportID;
    UInt32     bitField;
    UInt32     collection;
    HIDUsage    collectionUsage;
    HIDUsage    collectionUsagePage;
    Boolean     isRange;
    Boolean     isStringRange;
    Boolean     isDesignatorRange;
    Boolean     isAbsolute;
    {
    struct
    {
        HIDUsage    usageMin;
        HIDUsage    usageMax;
        UInt32     stringMin;
        UInt32     stringMax;
        UInt32     designatorMin;
        UInt32     designatorMax;
    } range;
    struct
    {
        HIDUsage    usage;
        HIDUsage    reserved1;
        UInt32     stringIndex;
        UInt32     reserved2;
        UInt32     designatorIndex;
        UInt32     reserved3;
    } notRange;
    }
};

```

HID Library Reference

```

        } u;
};
typedef HIDButtonCaps * HIDButtonCapsPtr;

```

The `HIDButtonCaps` structure is used by HID clients to hold the capabilities data for a button on a HID device.

HIDValueCaps

```

struct HIDValueCaps
{
    HIDUsage    usagePage;
    UInt32     reportID;
    UInt32     bitField;
    UInt32     collection;
    HIDUsage    collectionUsage;
    HIDUsage    collectionUsagePage;

    Boolean     isRange;
    Boolean     isStringRange;
    Boolean     isDesignatorRange;
    Boolean     isAbsolute;

    UInt32     bitSize;
    UInt32     reportCount;

    SInt32     logicalMin;
    SInt32     logicalMax;
    SInt32     physicalMin;
    SInt32     physicalMax;

    union
    {
        struct
        {
            HIDUsage    usageMin;
            HIDUsage    usageMax;
            UInt32     stringMin;
            UInt32     stringMax;
        }
    }
}

```

HID Library Reference

```

        UInt32    designatorMin;
        UInt32    designatorMax;
    } range;
    struct
    {
        HIDUsage  usage;
        HIDUsage  reserved1;
        UInt32    stringIndex;
        UInt32    reserved2;
        UInt32    designatorIndex;
        UInt32    reserved3;
    } notRange;
    } u;
};
typedef HIDValueCaps * HIDValueCapsPtr;

```

The `HIDValueCaps` structure is used by HID clients to hold the capabilities data for a value from a HID device.

HIDDeviceDispatchTable

This is the `HIDDeviceDispatchTable`. The `HIDDeviceDispatchTable` contains the entry points for the functions that HID drivers must implement. Your driver exports the symbol `TheHIDDeviceDispatchTable`. The documentation that describes how a HID driver implements the dispatch table APIs is found in “Communicating with the Human Interface Device Driver” in Chapter 4, “Writing Mac OS USB Drivers,” and in the `HID.h` file.

```

struct HIDDeviceDispatchTable {
    UInt32    dispatchTableCurrentVersion;
    UInt32    dispatchTableOldestVersion;
    UInt32    vendorID;
    UInt32    vendorSpecific;
    UInt32    reserved;

    HIDGetDeviceInfoProcPtr    pHIDGetDeviceInfo;
    HIDGetHIDDescriptorProcPtr    pHIDGetHIDDescriptor;
    HIDOpenDeviceProcPtr    pHIDOpenDevice;
    HIDCloseDeviceProcPtr    pHIDCloseDevice;
};

```

HID Library Reference

```

HIDInstallReportHandlerProcPtr    pHIDInstallReportHandler;
HIDRemoveReportHandlerProcPtr    pHIDRemoveReportHandler;
HIDCallPreviousReportHandlerProcPtr pHIDCallPreviousReportHandler;
HIDGetReportProcPtr              pHIDGetReport; /* new for vers 2, was */
                                   /* reserved in version 1*/
HIDSetReportProcPtr              pHIDSetReport; /* new for vers 2, was */
                                   /* reserved in version 1*/
HIDControlDeviceProcPtr          pHIDControlDevice;
};

```

HIS Library Error Codes

Table contains a list of error codes returned by the HID library. The HID library returns errors in the range -13949 to -13900.

Table 7-1 HID library error codes

Error code	Error name	Description
0	kHIDSuccess	
-13950	kHIDBaseError	
-13949	kHIDNullStateErr	
-13948	kHIDBufferTooSmallErr	
-13947	kHIDValueOutOfRangeErr	
-13946	kHIDUsageNotFoundErr	
-13945	kHIDNotValueArrayErr	
-13944	kHIDInvalidPreparsedDataErr	
-13943	kHIDIncompatibleReportErr	
-13942	kHIDBadLogPhysValuesErr	
-13941	kHIDInvalidReportTypeErr	
-13940	kHIDInvalidReportLengthErr	
-13939	kHIDNullPointerErr	

HID Library Reference

Table 7-1 HID library error codes

Error code	Error name	Description
-13938	kHIDBadParameterErr	
-13937	kHIDNotEnoughMemoryErr	
-13936	kHIDEndOfDescriptorErr	
-13935	kHIDUsagePageZeroErr	
-13934	kHIDBadLogicalMinimumErr	
-13933	kHIDBadLogicalMaximumErr	
-13932	kHIDInvertedLogicalRangeErr	
-13931	kHIDInvertedPhysicalRangeErr	
-13930	kHIDUnmatchedUsageRangeErr	
-13929	kHIDInvertedUsageRangeErr	
-13928	kHIDUnmatchedStringRangeErr	
-13927	kHIDUnmatchedDesignatorRangeErr	
-13926	kHIDReportSizeZeroErr	
-13925	kHIDReportCountZeroErr	
-13924	kHIDReportIDZeroErr	
-13923	kHIDInvalidRangePageErr	
-13910	kHIDDeviceNotReady	
-13909	kHIDVersionIncompatibleErr	

Changes In Mac OS USB Software

This appendix includes a general discussion of the features in version 1.1 of the Mac OS USB software. In addition, it defines the use of the version 1.1 USBP parameter block that supports isochronous transfers, and it includes some discussion of the feature enhancements in version 1.2 of the Mac OS USB software.

There are significant differences in the features supported in version 1.1 and later of the Mac OS USB software. To take advantage of the new features some modification of existing code that supported version 1.0 Mac OS USB software 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 226).

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 defined in “The USBPB Parameter Block” (page 94)
- Multiple bus support
- Improved bus enumeration
- Driver notification messages that support Mac OS sleep and wake
- Improved functionality for USB control requests

Changes In Mac OS USB Software

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 225) and “USB Software Presence and Version Attributes” (page 45), rather than the version number to ensure that the features you are interested in are supported on the Macintosh your software is running on.

Improved Bus Enumeration

Version 1.1 provides improved bus enumeration 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`

Changes In Mac OS USB Software

Message constant name

```
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 ('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 121) 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 specify the flag `kUSBAddressRequest` 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.

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
<code>usbBMRequestType</code>	<code>usb.cntl.BMRequestType</code>
<code>usbBRequest</code>	<code>usb.cntl.BRequest</code>
<code>usbWValue</code>	<code>usb.cntl.WValue</code>
<code>usbWIndex</code>	<code>usb.cntl.WIndex</code>

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

```
OSStatus USBDriverNotifyProc (
    USBDriverNotification notification,
    void *pointer,
    UInt32 refcon);
```

Changes In Mac OS USB Software

Drivers that were compiled with earlier versions of the USB.h header file will have a different `kUSBClassDriverPluginVersion` value and the USB Manager will call the `USBDriverNotifyProc` without the `refcon` parameter.

Check that the current version of USB software has isochronous support before making the `USBIsocRead` or `USBIsocWrite` calls. You must also weak link your class driver with the `USBServicesLib` USB.h file. If your driver makes `USBIsocRead` and `USBIsocWrite` calls and hard links to the `USBServicesLib` file, the system will check for support of these calls. If it finds that they are not available, the driver will not load.

Major Features Introduced In Version 1.2

Beginning with version 1.2 of the Mac OS USB software, multiple class drivers can be merged into a single Mac OS extension file. This feature is beneficial in situations where you have several vendor specific devices in a product family, each device requiring slightly different driver functionality. Rather than creating a standalone native driver for each device, you can write driver code for each device and merge all the driver code into a single file. The single file will be loaded based on vendor and product specific ID information in favor of the Apple generic drivers, which guarantees proper support your family of devices.

The steps below define how to merge USB class drivers into a single file using CodeWarrior IDE 2.0 or greater. This discussion assumes familiarity with the Macintosh and CodeWarrior programming environments.

1. Create each class driver into a Shared Library file of type 'shlb', not 'ndrv' as you presently would for a standalone USB class driver.
2. Set the CodeWarrior Target Setting with the linker popup to Mac OS Merge. Note that this target could be within the same project that creates the separate driver files to be merged.
3. In the Mac OS Merge Panel under the Linker grouping, set the Project type popup menu to Shared Library, set the file name and set the file creator to 'usbd', and file type to 'ndrv'. Ensure that the Copy Code Fragments option is checked. If there are resources associated with either of the drivers, then make sure that the Copy Resources option is checked.
4. For this target, set the source files to be the shared library files which contain the drivers to be merged.

Changes and Enhancements Introduced in Version 1.4

This section provides information about changes and enhancements introduced in version 1.4 since version 1.3.5 of the Mac OS USB software was released.

- Added feature for a USB Shim to be registered as a shared library. The USB Shim must export the symbol "ShimDescription" which is a structure of type `USBShimDescription` where the `USBShimLoadingOptions` field must have the `kUSBRegisterShimAsSharedLibrary` bit set, and the `libraryName` field defines the CFM shared library name.
- Added support for the `HIDDriver`, `HIDGetReport`, and `HIDSetReport` APIs.
- The OHCI UIM has been modified so that the root hub simulation is interrupt based instead of polled. Other changes were made to the USB keyboard and pointing devices to work with Macsbug.
- Fixed a buffer overrun situation with the USB Audio class driver which resulted in the corruption of the heap. The common symptom was that USB microphones would work for a period of time, but then input would stop.
- Implemented several fixes in the USB Audio class for compatibility with audio input and output devices.
- Fixed an error-recovery problem in the printer class driver.
- Modified USB Family Expert to report better error messages when a driver fragment fails to load.
- Modified the HID class driver for compatibility with USB 1.1. Version 1.1 of the Mac OS USB software is now the earliest version on which the Apple HID driver loads.
- Added support to the HID class driver for some digitizer tablets.
- Exported the `USBAddShimFromDisk` function from the USB Family Expert so that the call now works properly.
- USB shims and drivers can now be in either type 'usbcd' or 'usbs' files.
- Added function `USBSetPipePolicy` to allow changing pipe characteristics. The function will be defined in the future version of this document. Examples of characteristics that can be controlled are:

Changes In Mac OS USB Software

- Max number of bytes per frame. Allows isochronous devices to use less than the bandwidth specified in the endpoint.
- Max data per IRP. Basically the max that `usbReqCount` can be. This is currently not implemented.
- Max service interval. For interrupt endpoints, this is the polling interval.
- Added the `USBGetStringDescriptor` function to obtain the USB String Descriptor as follows:

```
OSStatus USBGetStringDescriptor(USBPB *pb)
```

The relevant fields in the USBPB for the `USBGetStringDescriptor` function are:

```
--> usbReference    Device/Interface
--> usb.cntl.WValue
                    String number
--> usb.cntl.WIndex
                    Language code, 0 for English
--> usbBuffer       Where to put descriptor
--> usbReqCount     Size of buffer
<-- usbActCount    Size of returned string
```

- Fixed a problem with suspend and resume so that transactions are processed correctly.
- Removed support in the Apple audio class driver for 4.000 kHz and 8.000 kHz sampling rates.
- Moved the USB.h file to the Interfaces folder. Added the HID.h file to the Interfaces folder.
- Added the `USBGetBandwidthAvailableByReference` function to obtain the bandwidth available to an isochronous device.

```
OSStatus USBGetBandwidthAvailableByReference
```

The parameters for the `USBGetBandwidthAvailableByReference` function are:

```
--> usbReference    device, interface. or pipe reference
<-- avail         currently available bandwidth for given usbReference
```

Changes In Mac OS USB Software

- Added the `kUSBDebugAwareFlag` to `USB.h`. This flag is used by drivers that want a transfer to complete while Macsbug is active. Typically users of this function will just be keyboard and mouse drivers. The use of this flag will be documented later. In order for Macsbug to work properly with drivers that implement this flag, the next version (later than that in Mac OS 9) of the MacOS ROM file is required.
- Fixed a problem with the Audio class driver where that the mouse was jerky with audio was active.
- Fixed a problem with the USB CardBus Enabler so that the UIM is unloaded before the card is ejected.
- The HID class driver checks for USB versions earlier than 1.1 and refuses to load.
- Addressed a crash problem with hot-plugging devices to a CMD 670/673 USB PCI card.
- Added the function `USBPortStatus`, defined as:

```
OSStatus USBPortStatus(USBPB *pb)
```

```
--> usbReference
```

Device or interface reference

- Included the draft version of the USB Software Locator technote, along with various tools for creating an Installer script and for testing the software update from the USB staging server.

Release Notes And Compatibility Issues

This section provides release notes and describes various Mac OS USB software implementation issues that developers should be aware of.

Bulk Data Transfer Performance Issues

As of Mac OS USB v1.2, an issue has been identified with how USB resources are used for USB bulk read and write calls for large data transfers. If a buffer is passed to a USB bulk call that is not aligned to a `MaxPacketSize` boundary, then USB may require up to twice as many Transfer Descriptor resources in order to

Changes In Mac OS USB Software

process the data coming in or going out. More importantly, for large transfer, passing a misaligned buffer means that the transport descriptor will be limited to a 4K transfer, while an aligned buffer extends the limit to 8K. For small transfers (less than 8K), this issue may not make a noticeable difference. For larger transfers, the use of a misaligned buffer can affect performance.

Note that the `MaxPacketSize` is the value read from the configuration descriptor. For bulk devices, the `MaxPacketSize` values are 1, 2, 4, 8, 16, 32 and 64.

Assuming that you want to pass in a buffer that is 4 times `MaxPacketSize`, the following algorithm demonstrates how one goes about setting up the `usbBuffer` that begins on a `MaxPacketSize` aligned address. Note that you are working with logical, not physical addresses.

```
#define kBufferMultiplier 4           // allocate a buffer that is 4x MaxPacketSize
UInt16    maxpacksize = 64;         // set for MaxPacketSize of
                                     // a bulk or control transaction

UInt8     *buffer;
UInt8     *alignedBuffer;

                                     // first allocate a buffer that is the desired
                                     // size + maxpacksize

buffer = NewPtrSys((kBufferMultiplier + 1) * maxpacksize);

                                     // figure out where the buffer falls on the
                                     // maxpacksize aligned address

alignedbuffer = (buffer + maxpacksize) & ~(maxpacksize - 1);
```

Following this example, you would set

```
usbReqCount = kBufferMultiplier * maxpacksize;
usbBuffer = alignedBuffer;
```

There is no limit as to the size of the buffer, however, the buffer size must be a multiple of `MaxPacketSize` for maximum performance.

Understanding Generic Drivers

Any driver that does not have the `kUSBDoNotMatchGeneric` flag set in the `usbDriverLoadingOptions` field in the `USBDriverDescription` structure is

Changes In Mac OS USB Software

technically considered a generic driver. Such a driver may be loaded for any device or interface that matches the class, subclass, and protocol specified in the `USBDriverDescriptor` structure.

Note however, that a driver is not really generic if it uses the `validateHWproc` function as defined in “USBClassDriverPlugInDispatchTable Structure” (page 65) to prevent it from being used for anything other than a specific set of vendor IDs and/or product IDs. Using the `validateHWProc` function is the best way for device developers to supply a generic style vendor-specific driver without utilizing multiple fragments in a single driver file. The intended purpose of the `ValidateHW` function was essentially to provide an alternative method for implementing generic-style driver functionality.

For developers that want to keep the list of valid vendor and product IDs in resources (or text files), loading those resource in your `ValidateHW` function, which is called at file-safe task time, would be the best way to do it.

IMPORTANT

Don't abuse the load generic ability. Always check for your device(s) in a `validateHW` function. The problems associated with not doing so are drivers that are matched to devices they know nothing about. For example, a printer driver that is loaded for the mouse because it matched device class 0/0 - composite, or a storage device driver that is loaded for a scanner (it could match interface class 0/0). Don't let this happen to you - use a `validateHW` function.

Core 99 CPU Issues

There are issues with the iBook, iMac DV, and the Power Macintosh G4 computer systems with AGP support USB and Mass Storage devices at startup. If media is present in a device when the system is started, the 1.3.3 driver in ROM is not replaced with a newer Mass Storage driver, if present. On the other hand, if media such as a Zip disk is inserted after the system has booted, the newer Mass Storage driver is loaded and used.

Audio Support in Version 1.3.5 and 1.4

USB audio support requires a Macintosh system with built-in USB ports and Mac OS 9.0. The Sound Manager components that are required to support USB

Changes In Mac OS USB Software

Audio devices are incorporated into the Mac OS ROM file, which is not loaded on systems that do not have built-in USB.

16 bit mono audio recording from USB audio input devices is not currently functional. A new audio software extension is required for this to work.

USB Extensions For Development and ROM Driver Conflicts

If you install the unmodified pre-release version 1.4x USB Support file available for software development along with the USB Device Extension file onto a system which has built-in USB, USB may fail. If you use the development extensions, they must function together. If the USB Support file is not activated, then the USB in ROM may try to use the USB device drivers out of the 1.4b1 USB Device Extension file, and may fail. In this case, you can reboot from the boot CD-ROM supplied with your Macintosh system, or you can reboot with Command-Option-ND key sequence held down which will keep the native drivers on disk from loading.

A P P E N D I X A

Changes In Mac OS USB Software

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

A P P E N D I X B

Conventions and Abbreviations

K	1024	μ s	microseconds
KB	kilobytes	ns	nanoseconds
kg	kilograms	Ω	ohms
kHz	kilohertz	sec.	seconds
k Ω	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

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.

USB Terminology

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.
device address	When used as a non-specific reference, a Universal Serial Bus device is either a hub or a function. 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.

USB Terminology

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 <i>whim</i> .
hub	A Universal Serial Bus device that provides additional attachment points to the Universal Serial Bus.

USB Terminology

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.

USB Terminology

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.

USB Terminology

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.

Index

A

abbreviations 235 to 236
aborting a pipe 136
ADB compatibility 23
ADB Manager 48
ADB shim 48
alternate interface 109
asynchronous calls 101

B

bandwidth 139
bulk transactions 128
bus busy errors 93
bus enumeration 224
bus errors 92
bus topology 30
byte ordering functions 152

C

cable length 19
`CallSecondaryInterruptHandler2` driver
 services function 103
CDM shim 48
class driver 42, 47
clearing a stall 137
closing an interface 119
code changes 226
communication flow 32
compatibility issues 23
compatibility shim 48, 73
completion routine 48, 102
completion routine execution context 102

configuration descriptor data structure 169
configuration descriptors 118
configuring device interfaces 113
connectors 19
constants
 descriptor type 167
 direction 165
 driver class 166
 driver loading 168
 endpoint 165
 error status level 168
 flag 164
 interface 166
 interface protocol 166
 pipe state 167
 power and bus attributes 167
 recipient 165
 type 165
 usbBRequest 165
control requests 124 to 126
control transactions 121
`CurrentExecutionLevel` driver services
 function 102
Cursor Device Manager 48

D

data exports 46
data structures
 HIDButtonCaps 218
 HIDCaps 215
 HIDCollectionNode 216
 HIDUsageAndPage 214
 HIDValueCaps 219
 USBConfigurationDescriptor 169
 USBDeviceDescriptor 169
 USBEndPointDescriptor 170

- USBHIDDescriptor 170
- USBHIDReportDesc 171
- USBInterfaceDescriptor 170
- USBPB 94
- data toggle synchronization 134
- data transfer types supported 24
- debugger aware flag 163
- default pipe 121, 134
- deprecated pipe functions 164
- descriptors 117
- descriptor type constants 167
- device
 - power 19
 - speed 19
- device access errors 91
- device callback request 187
- device configuration 107
- device descriptor data structure 169
- device detection 176
- device detection example 83
- device endpoint 117
- device examples 20
- device notification 84, 144
- device notification callback routine 185
- device notification parameter block 186
- device notifications 187
- device power requirements 109
- device reference 94
- device removal notification 145
- device requests 120 to 123
- device reset 142
- devices 33
- direction constants 165
- driver class constants 166
- driver descriptor structure 48
- driver dispatch table 48
- driver file and resource type 168
- driver initialization routine 48
- driver loading 187
- driver loading options 168
- driver logic errors 93
- driver matching 47
- driver notification messages 224
- drivers, loading after boot time 184

E

- endpoint 0 34
- endpoint constants 165
- endpoint descriptor data structure 170
- endpoints 33
- errors
 - bus 92
 - bus busy 93
 - device access 91
 - driver logic 93
 - incorrect command 93
 - overrun 93
 - parameter block 99
 - underrun 93
- error status level constants 168
- expansion capabilities 18

F

- file types 168
- finding an interface 107 to 110
- flag constants 164
- frActCount field 100
- FrameList field 101
- frames, isochronous transfers 99
- frReqCount field 100
- frStatus field 100
- functions
 - USBAbortPipeByReference 136
 - USBAddDriverForFSSpec 184
 - USBAddShimFromDisk 183
 - USBAllocMem 150
 - USBBulkRead 128
 - USBBulkWrite 130
 - USBClearPipeStallByReference 137
 - USBClosePipeByReference (deprecated) 164
 - USBConfigureInterface 113
 - USBControlRequest 124
 - USBDeallocMem 151
 - USBDelay 147
 - USBDeviceRequest 121
 - USBDisposeInterfaceRef 119

- USBDriverNotify 182
- USBExpertFatalError 157
- USBExpertGetStatusLevel 156
- USBExpertInstallDeviceDriver 144
- USBExpertInstallInterfaceDriver 145
- USBExpertNotifyParent 183
- USBExpertRemoveDeviceDriver 145
- USBExpertRemoveInterfaceDriver 146
- USBExpertSetStatusLevel 156
- USBExpertStatus 154
- USBFindNextAssociatedDescriptor 117
- USBFindNextEndpointDescriptorImmediate 1
61
- USBFindNextInterface 108
- USBFindNextInterfaceDescriptorImmediate 15
9
- USBFindNextPipe 115
- USBGetBandwidthAvailableByReference 139
- USBGetConfigurationDescriptor 118
- USBGetFrameNumberImmediate 148
- USBGetFullConfigurationDescriptor 158
- USBGetPipeStateByReference 135, 139, 141
- USBGetPipeStatusByReference 135
- USBGetVersion 106
- USBIntRead 127
- USBIntWrite 127
- USBIsocRead 131
- USBIsocWrite 132
- USBMakeBMRequestType 121
- USBNewInterfaceRef 112
- USBOpenDevice 110
- USBOpenPipe (deprecated) 164
- USBPortStatus 143
- USBResetDevice 142
- USBResetPipeByReference 136
- USBResumeDeviceByReference 143
- USBSetConfiguration 110
- USBSetPipeActiveByReference 138
- USBSetPipeIdleByReference 138
- USBSetPipePolicy 140
- USBSuspendDevice 142

G

- gestalt selectors 23
- gestaltUSBHasIsoch selector 225
- getting the pipe state 135

H

- HIDButtonCaps data structure 218
- HIDCallPreviousReportHandlerProcPtr dispatch
table entry 79
- HIDCaps data structure 215
- HIDCloseDeviceProcPtr dispatch table entry 78
- HIDCollectionNode data structure 216
- HIDControlDeviceProcPtr dispatch table
entry 80
- HID descriptor data structure 170
- HIDDeviceDispatchTable structure 76
- HID device info selectors 77
- HID dispatch table entries
 - HIDCallPreviousReportHandlerProcPtr 79
 - HIDCloseDeviceProcPtr 78
 - HIDControlDeviceProcPtr 80
 - HIDGetDeviceInfoProcPtr 77
 - HIDGetHIDDescriptorProcPtr 77
 - HIDGetReportProcPtr 79
 - HIDInstallReportHandlerProcPtr 79
 - HIDOpenDeviceProcPtr 78
 - HIDRemoveReportHandlerProcPtr 79
 - HIDReportHandlerProcPtr 78
 - HIDSetReportProcPtr 80
- HID dispatch table symbol 83
- HIDGetDeviceInfoProcPtr dispatch table
entry 77
- HIDGetHIDDescriptorProcPtr dispatch table
entry 77
- HIDGetReportProcPtr dispatch table entry 79
- HIDInstallReportHandlerProcPtr dispatch table
entry 79
- HID library API 190
- HID library constants 212
- HID library data structures 214
- HID library error codes 220 to 221

HID library functions

- HIDCloseReportDescriptor 193
- HIDGetButtonCaps 194
- HIDGetButtons 200
- HIDGetButtonsOnPage 202
- HIDGetCaps 194
- HIDGetCollectionNodes 195
- HIDGetScaledUsageValue 203
- HIDGetSpecificButtonCaps 196
- HIDGetSpecificValueCaps 197
- HIDGetUsageValue 204
- HIDGetUsageValueArray 205
- HIDGetValueCaps 199
- HIDMaxUsageListLength 199
- HIDOpenReportDescriptor 192
- HIDSetButton 207
- HIDSetButtons 208
- HIDSetSelectUsageValue 209
- HIDSetUsageValue 210
- HIDSetUsageValueArray 211
- HIDUsageListDifference 206

HID module device dispatch table 81

HIDOpenDeviceProcPtr dispatch table entry 78

HIDRemoveReportHandlerProcPtr dispatch table entry 79

HID report descriptor data structure 171

HIDReportHandlerProcPtr dispatch table entry 78

HIDSetReportProcPtr dispatch table entry 80

HIDUsageAndPage data structure 214

HIDValueCaps data structure 219

high-speed device 21

holding data buffer memory 124

host software 30

HostToUSBLong function 153

HostToUSBWord function 153

hub device 22

hub driver 47

I

incorrect command errors 93

interface 33, 107

- alternate 109
 - closing 119
- interface constants 166
- interface descriptor 117
- interface descriptor data structure 170
- interface protocol constants 166
- interface reference 48
- interrupt data transfers 126, 127
- introduction to USB 18
- isochronous calls 99
- isochronous parameter block 226
 - frActCount field 100
 - FrameList field 101
 - frReqCount field 100
 - frStatus field 100
 - NumFrames field 101
 - packet error code 100
 - usbBuffer field 100
 - usbFrame field 101
 - usbReference field 100
 - usbReqCount field 100
 - usbStatus field 100
- isochronous pipes 101

K

keyboards supported 24

kNotifyUSBSysSleepDemand 225

kNotifyUSBSysSleepRequest 224

kNotifyUSBSysSleepRevoke 225

kNotifyUSBSysSleepWakeUp 225

kUSBCompletionError 99

kUSBDebuggerAwareFlag usage 163

kUSBDontMatchGenericDevice constant 168

kUSBDontMatchInterface constant 168

kUSBFlagsError 99

kUSBInterfaceMatchOnly constant 168

kUSBPBLengthError 99

kUSBPBVersionError 99

kUSBProtocolMustMatch constant 168

L

loading an interface driver 145
 logical topology 31
 low-speed device 21
 low-speed device cables 19

M

maximum packet size 93
 maxpacketsize 93
 MaxPacketSize value 129
 memory functions 150 to 152
 merging code fragments 227
 multiple bus support 148
 multiple device support 26, 227
 mutiple USB controllers 224

N

Name Registry 46
 ndrvc code fragment 46
 network compatibility 23
 no data timeout 105
 non-0 endpoints 34
 non-asynchronous calls 101
 NumFrames field 101

O

OLDBUSNAMES macro 226
 OpenFirmware 46
 opening a device 110
 opening an interface 112
 over run errors 93

P

packet 93
 packet size 93
 parameter block errors 99
 physical topology 31
 pipe descriptor 117
 pipes 34
 pipe stall 134
 pipe state constants 167
 pipe state control functions 133
 polling fields 102
 port status 143
 power and bus attribute constants 167
 power features 22
 power management features 224

R

recipient constants 165
 references 93
 removing an interface driver 146
 reset device 142
 resetting a device 141
 resetting a pipe 136
 resource types 168
 resuming a device 143
 root hub 22

S

secondary interrupt level 102
 setting a pipe active 138
 setting a pipe to idle 138
 setting the configuration 110
 shim 48, 73
 shim, registering 183
 sleep notification messages 224
 storage devices 24
 suspending a device 142
 system task level 102

T

time functions 146 to 149
topology database access functions 177
transaction functions 124 to 130
transaction timeout 105
transaction timeout error 105

U

UIM 42, 45
underrun errors 93
USB
 bus topology 30
 communication flow 32
 compatibility issues 23
 connectors 19
 device class examples 20
 device expansion 18
 devices 20, 33
 endpoint 0 34
 endpoints 33
 gestalt selectors 23
 high-speed device 21
 host software 30
 hub devices 22
 interface 33
 introduction to 18
 logical topology 31
 low-speed device 21
 network compatibility 23
 non-0 endpoints 34
 parameter block 94
 physical topology 31
 pipes 34
 power features 22
 root hub 22
 storage devices 24
 supported controllers 25
 supported data transfer types 24
 supported keyboards 24
USBAbortPipeByReference function 136
USBAddDriverForFSSpec function 184

USBAddShimFromDisk function 183
USBAllocMem function 150
usbBMRequest
 direction constants 165
 recipient constants 165
 type constants 165
usbBRequest constants 165
usbBuffer field 100, 124
USBBulkRead function 128
USBBulkWrite function 130
USBClassDriverPlugInDispatchTable
 structure 226
USB class drivers 47
USBClearPipeStallByReference function 137
USBClosePipeByReference function
 (deprecated) 164
USBConfigurationDescriptor data structure 169
USB configuration services 107 to 120
USBConfigureInterface function 113
USB constants 164
USB controllers supported 25
USBControlRequest function 124, 124 to 126
USB_CONTSTANT16 macro 152, 153
USBDeallocMem function 151
USBDelay function 147
USBDeviceDescriptor data structure 169
USBDeviceNotificationCallbackProc
 function 185
USBDeviceRefToBusRef function 181
USBDeviceRequest function 121 to 123
USBDisposeInterfaceRef function 119
USBDriverNotificationProcPtr sleep notification
 messages 224
USBDriverNotify function 182
USBDriverNotifyProcPtr prototype 226
USBEndPointDescriptor data structure 170
USBExpertFatalError function 157
USBExpertGetStatusLevel function 156
USBExpertInstallDeviceDriver function 144
USBExpertInstallInterfaceDriver function 145
USBExpertNotifyParent function 183
USBExpertRemoveDeviceDriver function 145
USBExpertRemoveInterfaceDriver function 146
USBExpertSetStatusLevel function 156
USBExpertStatus function 154

- USBFindNextAssociatedDescriptor function 117
- USBFindNextEndpointDescriptorImmediate function 161
- USBFindNextInterfaceDescriptorImmediate function 159
- USBFindNextInterface function 108
- USBFindNextPipe function 115
- usbFrame field 101
- USB frames 146
- USB Gestalt selectores 45
- USBGetBandwidthAvailableByReference function 139
- USBGetConfigurationDescriptor function 118
- USBGetDeviceDescriptor function 178
- USBGetDriverConnectionID function 181
- USBGetFrameNumberImmediate function 148
- USBGetFullConfigurationDescriptor function 158
- USBGetInterfaceDescriptor function 178
- USBGetNextDeviceByClass function 179
- USBGetPipeStateByReference function 135, 139, 141
- USBGetPipeStatusByReference function 135
- USBGetVersion function 106
- USBHIDDescriptor data structure 170
- USBHIDDeviceDispatchTable 83
- USBHIDModuleDispatchTable 81
- USBHIDReportDesc data structure 171
- USB hub driver 47
- USBInterfaceDescriptor data structure 170
- USB Interface Module 42, 45
- USBIntRead function 127
- USBIntWrite function 127
- USBIsocFrame structure 100
- USBIsocRead function 131
- USBIsocWrite function 132
- USBMakeBMRequestType function 121
- USB Manager 42, 46, 176
 - APIs 177
 - device notification parameter block 186
 - getting device descriptors 178
 - getting driver connection ID 181
 - getting drivers by class 179
 - getting interface descriptors 178
 - getting the device bus reference 181
 - responsibilities 177
 - topology database access functions 177
- USB Manager data structures
 - USBDeviceNotificationParameterBlock 186
- USB Manager functions
 - USBAddDriverForFSSpec 184
 - USBAddShimFromDisk 183
 - USBDeviceNotificationCallbackProc 185
 - USBDeviceRefToBusRef 181
 - USBGetDeviceDescriptor 178
 - USBGetDriverConnectionID 181
 - USBGetInterfaceDescriptor 178
 - USBGetNextDeviceByClass 179
 - USBInstallDeviceNotification 187
 - USBRemoveDeviceNotification 188
- USBNewInterfaceRef function 112
- USBOpenDevice function 110
- USBOpenPipe function (deprecated) 164
- USBPB, required fields 98
- USBPB parameter block 94
- USBPB pbLength field 98
- USBPB pbVersion field 98
- USBPB usbBMRequestType field 98
- USBPB usbCompletion field 98
- USBPB usbFlags field 98
- USBPB usbRefcon field 98
- USBPortStatus function 143
- USB Prober application 35
- USB Prober features 37
- usbReference field 100
- USB reference types 93
- usbReqCount field 100
- USBResetDevice function 142
- USBResetPipeByReference function 136
- USBResumeDeviceByReference function 143
- USB Services Library 42, 48, 91
- USBSetConfiguration function 110
- USBSetPipeActiveByReference function 138
- USBSetPipeIdleByReference function 138
- USBSetPipePolicy function 140
- USB software architecture 44
- USB software components 42
- usbStatus field 100
- USBsuspendDevice function 142
- USBToHostLong function 154

I N D E X

- USBToHostWord function 153
- USB topology 30
- USB transaction functions
 - setting up data buffer 124
- usbWIndex field 110
- usbWValue field 110
- USL 42, 48, 91
- USL data structures 168
- USL error reporting 91
- USL logging services 154 to 157
- USL USB management services 144
- USL utility functions 146

V

- vendor specific device support 26
- version 1.1 USB software 223
 - features 223
 - USB.h file 226
- version 1.2 USB software 26, 227
- virtual memory 124

I N D E X

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.