

THIS SPEC IS OBSOLETE

Spec No: 001-65423

Spec Title: MAC OS X: GETTING STARTED WITH USB - AN1105

Sunset Owner: Gayathri Vasudevan (GAYA)

Replaced by: None

Mac OS X: Getting Started with USB

AN1105

Associated Project: No
Associated Application Notes: None

Abstract

Developing USB drivers for Mac OS X is completely different than developing USB drivers on Mac OS 9. This application notes describe how to develop USB driver for Mac OS X. Include introducing the kernel of Mac OS X and driver architecture. Some useful example codes also be attached.

Introduction

This document discusses developing USB drivers for Mac OS X. We will briefly discuss the USB system architecture on Mac OS X to understand the terminology. Then we will discuss issues and tips and tricks for developing USB device drivers.

Specifically we want to cover the following:

1. Mac OS X Kernel and I/O Kit architecture
2. USB on Mac OS X
3. Kernel Space drivers
4. User Space drivers
5. Interaction with the Classic environment
6. Reference information and where to go next

Comparing Mac OS 9 to Mac OS X

Developing USB drivers for Mac OS X is completely different than developing USB drivers on Mac OS 9. It is unfortunate, but the skills used to develop drivers for Mac OS 9 are not transferable to Mac OS X. The API's are different. The runtime environment is different and the tools are different. The table in Figure 1, compares some of the differences between USB drivers on Mac OS 9 versus Mac OS X.

Figure 1. USB Drivers on Mac OS 9 vs. Mac OS X

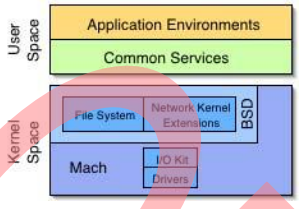
	Mac OS 8 & 9	Mac OS X
Driver Description	In driver's code fragment	Property List (XML file)
Scheduling	Asynchronous	Asynchronous & Synchronous
Programming	C	Embedded C++ (object oriented)
Driver Framework	USB Specific	I/O Kit
Providing Application Access	Easy – Direct Access	Hard – Must bridge Kernel Boundary!
Similar function names/methods	Parameter Block interface	Series of C++ methods
Memory buffers	Held and locked	IOMemoryDescriptor

OBSOLETE

Mac OS X Architecture Overview

In the simplest form, the Mac OS X architecture can be divided into two parts: user space and kernel space. The diagram commonly used to describe the Mac OS X architecture is shown in Figure 2.

Figure 2. Mac OS X Architecture Overview



The lower block, in Figure 2, represents the kernel. All code that runs in the kernel shares the kernel's address space. Everything else outside of the kernel is referred to as user space. Applications built with the Carbon, Cocoa, Java or BSD application environments get their own address space at runtime, but by convention it's all referred to as user space. Common services used by the application environments, including the Event Manger and the Quartz imaging and windowing system, are also in user space.

The Mac OS X Kernel is based upon Mach and FreeBSD and is available via open source licensing through the Darwin project (http://www.apple.com/darwin). In other operating systems it's common to refer to just the Mach Kernel as the kernel. However, Apple refers to everything you see in the lower block as the Kernel.

Inside the kernel is the I/O Kit – Apple's object-oriented framework for developing KEXTs (a.k.a, Kernel Extensions or kernel drivers). The I/O Kit is based upon a restricted form of C++ that provides an inheritance model to make writing drivers easier. The I/O Kit enables the rapid development of device drivers by implementing abstractions common to all drivers as well as abstractions that are specific to certain types of drivers (such as USB and FireWire).

The I/O Kit framework provides mechanisms to handle I/O synchronization, memory management, as well as object runtime and life cycle support. The kernels used in operating systems such as Microsoft NT, Linux and most Unix derivations are single threaded, which simplifies kernel driver development. However, the Mach Kernel in Mac OS X is multi-threaded, which requires drivers to adopt special synchronization mechanisms provided by I/O Kit.

The I/O Kit also provides a plug-in mechanism, called Device Interfaces. The Device Interface plug-in mechanism enables applications and user-space drivers to access hardware. Or, more specifically, it allows applications and user-space drivers to access KEXTs that drive hardware. We will discuss Device Interfaces more later when we discuss user-space drivers.

USB on Mac OS X

The USB system software had to be completely redesigned for Mac OS X based upon the I/O Kit. The USB system software on Mac OS X is referred to as the IOUSBFamily. The IOUSBFamily is deployed as a single KEXT bundle. We will describe what a KEXT bundle is in the section Anatomy of a KEXT. The IOUSBFamily is a required part of Mac OS X. The IOUSBFamily KEXT can be found in the directory /System/Library/Extensions/IOUSBFamily.kext.

The IOUSBFamily provides support for many USB Class device drivers in Mac OS X:

- Hub driver
- HID devices – keyboards, mice, gaming, pointing devices, UPSes
- Mass Storage (including optical storage like CD-R/W and DVD)
- USB Communication Class (V.90/V.25 Modems)
- Audio – Audio input and output
- Printing – Built-in PostScript printing support

Highlights of USB and Mac OS X

Kernel vs. User space. Even though all of IOUSBFamily is in the Kernel, Mac OS X drivers can be written in kernel space or user space and have direct access to the IOUSBFamily. Applications and user-space drivers can take full control over a USB device using the device interface plug-in mechanisms provided by IOUSBFamily and I/O Kit.

I/O Kit and C++. I/O Kit provides an object oriented framework for developing drivers. It is implemented in a restricted form of C++ that omits features unsuitable for use within a multithreaded kernel (e.g., RTTI, exceptions, etc). The IOUSBFamily and I/O Kit enable the rapid development of USB device drivers by implementing abstractions common to all USB drivers. IOUSBFamily objects can be instantiated and used directly or they can serve as base classes from which custom drivers can be derived and functionality can be overridden. The I/O Kit's object oriented inheritance model and the IOUSBFamily eliminate a lot of the USB specific code that a USB driver would have to implement. We will provide examples of the inheritance model in action in the section on Kernel Space Drivers.

Driver Matching. One of the unique aspects of Mac OS X's driver model has to do with how driver matching and loading works. I/O Kit uses matching dictionaries, encoded in KEXT property lists, to match device drivers to hardware. Applications and user-space drivers create matching dictionaries used to find and access hardware. Because driver matching is very important, we will discuss it further in this section as well as in the Developing Kernel Space Drivers section and the Developing User Space Drivers section. Unlike in Mac OS 9, drivers are not loaded automatically simply because they are installed. In Mac OS X, a driver must first be matched to an existing device before that driver can be loaded.

I/O Registry. The I/O Registry is a dynamic database representing a collection of objects (nubs and drivers) and the provider-client relationships between the objects. The I/O Registry exists only in memory and is rebuilt every time the system boots. The I/O Registry dynamically changes as new hardware is added or removed from the system. The I/O Kit provides powerful search APIs for searching the I/O Registry for an object with particular characteristics. This is how applications find the Device Interfaces that are used to access hardware devices.

CONFIDENTIAL

Blocking I/O. Any I/O done in Mac OS 9, regardless if from applications or drivers, had to be called asynchronously. However, in Mac OS X asynchronous and synchronous I/O is possible. Synchronous (blocking) I/O simplifies application code that performs I/O. An application or user-space driver can make an I/O call and the execution of that code will block until the I/O completes. Asynchronous I/O is also available for the more advanced applications or user-space drivers.

Multithreaded Drivers. The Mac OS X kernel is multithreaded. The kernels used in operating systems such as Microsoft NT, Linux and most Unix derivations are single threaded. This means that Mac OS X kernel drivers must be able to handle multithreading. Multithreaded driver support is another important feature provided by I/O Kit.

Memory Buffer Comparisons

In Mac OS 9, applications create memory buffers that need to be marked as "held" memory. This means the memory buffers cannot be swapped out to disk by the virtual memory system. Then pointers to the memory buffers are passed to drivers.

In Mac OS X, user-space drivers use virtual address buffers. They pass memory pointers straight into the kernel. But in the kernel the memory pointer is wrapped inside an IOMemoryDescriptors object. IOMemoryDescriptors are very flexible. The memory buffers IOMemoryDescriptors reference do not need to be contiguous. The user-space memory buffers passed into the kernel are not necessarily mapped into kernel address space. IOMemoryDescriptors objects can describe memory that is inaccessible from the kernel directly, but is accessible using DMA. This is very efficient because wiring memory into kernel address space would be a very expensive operation.

Checking out Darwin and checking out the IOUSBFamily

The IOUSBFamily is part of the Darwin open source project. All of the source code to the IOUSBFamily is available via open-source licensing. The URL is cvs.opensource.apple.com:/cvs/Darwin. Download the source to IOUSBFamily and get hands on understanding how USB works on Mac OS X. The major objects in the IOUSBFamily are listed here with a brief summary.

IOUSBController object – Heart of USB Family. The AppleOHCI driver inherits from this object to provide access to the USB Host Controller shipped with Mac hardware.

IOUSBDevice object – Its provider is the IOUSBController object. There is one IOUSBDevice object per device connected to the bus. Provides methods for accessing the fields of the USB Device Descriptor and the Configuration Descriptor of the device. It sets the configuration of the device and instantiates an IOUSBInterface object for each interface in the configuration descriptor.

IOUSBInterface Class – Its provider is the IOUSBDevice object. There is one IOUSBInterface object per interface on a device – Interface Descriptor. Instantiates and distributes IOUSBPipe objects to drivers. Allows setting of Alternate interfaces.

IOUSBPipe Class – There is one IOUSBPipe object per endpoint in an Interface Descriptor. The IOUSBPipe provides the methods for data transfer to drivers. IOUSBPipe objects provide access to all the USB Endpoints:

- Control
- Interrupt Read/Write
- Bulk Read/Write
- Isochronous Read/Write

IOUSBDeviceUserClient and IOUSBInterfaceUserClient – Provide connections between user tasks and the IOUSBDevice and the IOUSBInterface kernel objects. The UserClient objects are visiable in the I/O Registry.

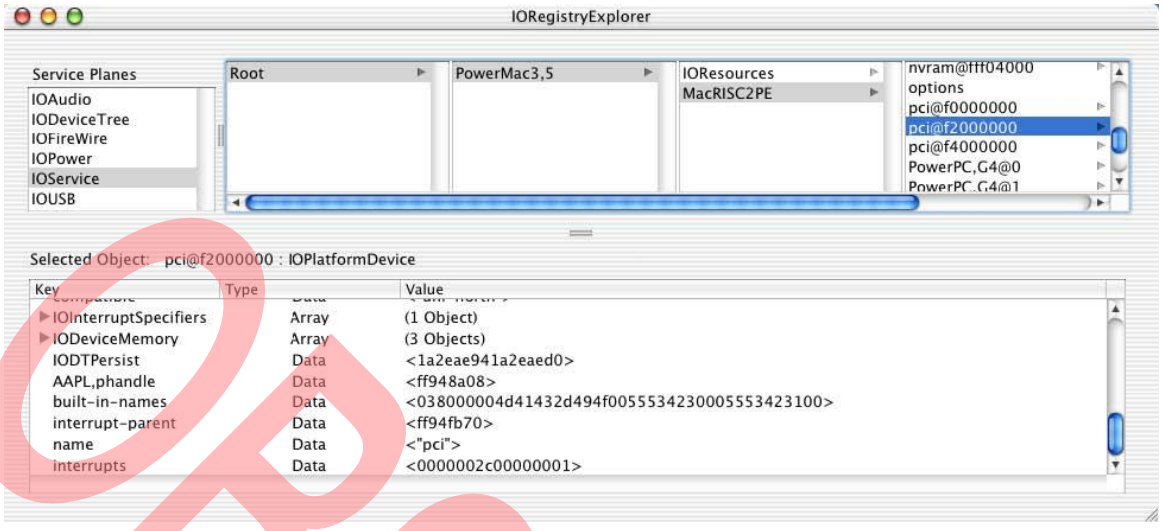
I/O Kit IOUSBFamily Kernel Driver Stack

In the previous section we covered the main objects in the IOUSBFamily. Let's take a look at how objects are built up in the system. This is also a good opportunity to experiment with the IORegistryExplorer utility. Open Developer/Applications/IORegistryExplorer.app.

The IORegistryExplorer allows the objects in the I/O Registry to be viewed from a number of different perspectives, referred to as Planes. All of the Planes start out with a root and build up from there. An IOUSB Plane exists to look at the objects in the I/O Registry from the perspective of the USB subsystem. The IOService Plane shows the provider-client relationships between the objects in the I/O Registry. The IOService Plane is the most useful Plane in the IORegistryExplorer, but it can also be most difficult to navigate. Navigating the first three levels of the IOService Plane can be the most confusing, an example has been captured in Figure 3.

PROOF

Figure 3. IORegistryExplorer Application



Select the IOService plane in the list. Then starting with the Root object, select the objects in the plane until you can select the pci@f2000000:IOPlatformDevice object, as shown in Figure 3. Then select the AppleMacRiscPCI object. With the AppleMac-RiscPCI object selected, look for the IOProviderClass key in the table at the bottom of the IORegistryExplorer window. This table displays the dictionary values stored in the I/O Registry for the selected object. The IOProviderClass is a key/value pair from a KEXT's property list. The IOProviderClass key/value pair tells I/O Kit what objects this KEXT can match to. In this case the AppleMacRiscPCI object matches to an object of type IOPlatformDevice.

Continuing from the AppleMacRiscPCI object, select the IOP-CIDevice object. The IOPCIDevice object is created when the PCI controller detects a USB controller chipset at boot time. Then, the IOUSBController class is matched to the IOPCIDevice object and instantiated. The IOUSBController object initializes the USB controller chip and then instantiates an IOUSBDevice object. The IOUSBDevice object represents the root hub, inside the controller chip. The IOUSBDevice object retrieves the USB device descriptor from the device and puts it into the I/O Registry to be used by I/O Kit to match and load the AppleUSBHubDriver. During initialization, the AppleUSBHubDriver scans the bus looking for USB devices. For each USB device found, an IOUSBDevice object is instantiated and the process continues recursively.

If we have USB Speakers attached, for example, the AppleUSBHubDriver tells the IOUSBController object to instantiate another IOUSBDevice nub object representing the speakers. The I/O Kit matches the IOUSBComposite class driver against this device because the USB Speakers are a composite class device. The IOUSBComposite driver sets the configuration in the device, which causes the interfaces to appear; the IOUSBComposite object calls the SetConfiguration method on IOUSBDevice and IOUSBDevice instantiates the interfaces of the device in the form of the IOUSBInterface nub objects. One of these interfaces is the audio channel. I/O Kit matches a driver against the IOUSBInterface nub object for the audio channel and in this case it is the AppleUSBAudioDevice driver. The driver stack is complete.

USB Driver Matching in Mac OS X

There are two kinds of objects found in the I/O Registry: nubs and drivers. The primary function of nubs is to provide matching services – matching drivers to devices. Driver matching is an I/O Kit process, so USB driver matching must follow I/O Kit rules. To support driver matching, each KEXT defines one or more IOKitPersonalities dictionaries that specify the kinds of devices it can support. An IOKitPersonalities dictionary is defined in the Info.plist property list in the KEXT's bundle. The dictionary values are used by I/O Kit to identify candidate drivers for a particular device.

When a USB device is attached to the bus, the hub driver detects and "enumerates" the device. The hub driver tells the USB controller driver to create an IOUSBDevice object. The IOUSBDevice object is an I/O Kit nub abstraction of the device's USB device descriptor. This IOUSBDevice object is attached to the IOService plane of the I/O Registry as a child of the USB controller driver. The IOUSBDevice nub object is then registered for matching with the I/O Kit.

I/O Kit finds and loads a driver for the nub in three distinct phases, using a subtractive process until a successful candidate is found. The phases of matching are: class matching, passive matching, and active matching.

Class matching – starts with all known KEXTs installed in the system and eliminates drivers of the wrong class. I/O Kit keeps track of all installed KEXTs in the file /System/Library/Extensions.mkext. I/O Kit looks for the nub's class name in the IOProviderClass key/value pair in each KEXT's property list. The IOProviderClass key/value pair tells I/O Kit what objects this KEXT can match to. All the KEXTs that match to this class remain in the list, all others are eliminated.

Passive matching – examines each remaining driver's personalities for properties specific to the device, eliminating those drivers that do not match. I/O Kit creates a table of possible USB drivers for the device using the information in the kernel drivers' property list. This table is then passed to the IOUSBFamily to evaluate the candidate drivers using a scoring mechanism based upon criteria specified by the *USB Common Class Specification*. This creates an ordered list of drivers with a ranking.

OBSOLETE

Active matching – begins when IOService calls the probe method for each candidate driver in the list starting with the highest ranked driver. The probe method is passed a reference to the provider object the driver is being matched against. The probe method can use the reference to the provider to verify that it can support the device. In additions, a probe score can be returned from the probe method to indicate how well suited the driver is to drive the device. However, for USB drivers it's usually best not to modify this value.

A driver can accept the USB device by returning a true value from the probe method. Once a driver accepts the USB Device, I/O Kit calls the driver's start method. At this time the driver can perform I/O to interrogate the device and determine if the driver really wants to support the device. If the driver changes it's mind and decides it actually doesn't support the device, it fails the start method and I/O Kit will then select the next candidate driver in the list and calls that driver's probe method.

Kernel vs. User Space Drivers

The first step to USB driver development in Mac OS X is determining if you are going to need to develop a kernel-space driver or a user-space driver. Some of the questions you should ask yourself are the following:

1. Will your driver be used by the kernel?
2. Will your driver be used by many tasks in the system?
3. Do you need access to primary interrupts?
4. Do you need tight synchronization between tasks?

If you answer yes to any of these questions then you might need to develop a kernel-space driver (a KEXT). If your driver needs to respond to PCI bus interrupts then you definitely need to be in the kernel. If your driver's client also lives in the kernel (e.g., storage drivers) then you need to be in the kernel. Kernel threads run at a higher priority than user-space threads. So, if you need very, very, tight synchronization between tasks then you will need to be in the kernel. If the previous questions don't apply to you, you should make it your goal to stay out of the kernel.

Debugging a KEXT is very difficult. Debugging a KEXT requires two-machine debugging using GDB (the gnu debugger). A single bug in a KEXT can bring down the whole system. Whereas user-space drivers can be developed using a source level debugger and crashes in user space won't bring down the system.

Recently, I met a developer who did all his preliminary USB driver development and prototyping in user space, because it's so much easier to do development. Then, only after the driver logic was fully debugged, did he move the code into a KEXT. This made his life easier.

Concrete Driver Examples

Keyboards and Mice – are kernel-space drivers because the HID System is in the kernel. And because they are used by just about every user-space task in the system.

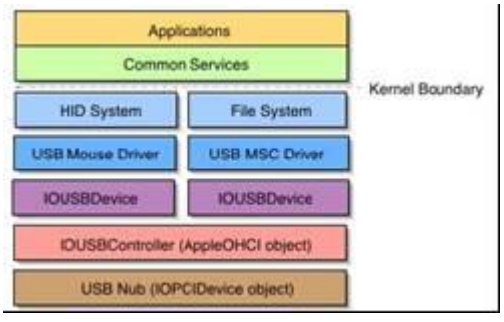
Mass Storage – are kernel-space drivers because they are used by the file system which is also in the kernel, see Figure 4.

Scanners – are user-space drivers because they will only be used by applications.

Printers – are user-space drivers because they will only be used by applications.

CD-R/W & DVD-R/W – are both kernel and user-space drivers due to sharing issues.

Figure 4. Directory Structure of a KEXT



USB Kernel Driver Development

A kernel-space driver is a Kernel Extension (KEXT) that is installed in /System/Library/Extensions. Currently, KEXTs must be built using Apple's Project Builder and debugged using GDB and command line tools in the Terminal. KEXTs must run in supervisor mode, and as such, they must have the appropriate user and permission settings. The KEXT and all it's contents must be owned by root and by the group wheel. Root and wheel are unix-isms.

Additionally the bundle has to have the file permissions of 755 and the files in the bundle need to have the permissions of 644. This can be performed with the following commands in the Terminal.

```
chown -R root your_driver.kext chown -R
:wheel your_driver.kext chmod -R 755
your_driver.kext
```

Anatomy of a KEXT

A KEXT is just a special kind of bundle. And a bundle is just a special kind of directory that the Finder treats as single file. Instead of a multi-forked file system with resource and data forks as was used in previous versions of the Mac OS, Mac OS X stores executable code and the software resources related to that code in a bundle. The bundle directory, in essence, "bundles" a set of resources into a discrete package.

But for a bundle to be a KEXT it must have the following:

1. The bundles name must end with a .kext extension.
2. All KEXTs must have an Info.plist – information property list (in XML) that describes the KEXT's contents and requirements, like a table of contents.

Optionally a KEXT can also contain:

1. The module, or KMOD, contains the Mach-O binary code. This is what's actually loaded into the kernel and run.
2. A Resources directory can be used for localized strings, icon, etc.
3. A Plug-ins directory can be used to hold a suite of drivers (e.g., IOUSBFamily.kext).

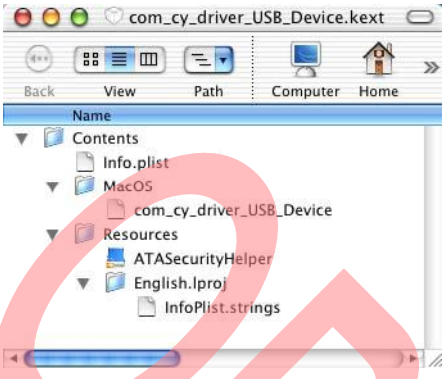
Normally, a KEXT has a module, but it can have none (locally). If a KEXT does not have a module, its property list must reference a module in another KEXT. The Info.plist

OBSOLETE

must refer to at least one module.

The contents of the KEXT bundle of the Cypress USB Mass Storage driver is shown in Figure 5.

Figure 5. Directory Structure of a KEXT



Inside of all bundles, exists a Contents directory where everything else is placed. The com_cy_driver_USB_device KMOD is inside a MacOS directory. Inside the Resources directory is an English localization folder and an ATASecurityHelper.app. ATASecurityHelper is a user-space application used by the Cypress USB Mass Stroage driver to provide a user interface to the ATA Security Feature for hard drives. This shows that just about anything can be stored in the Resources directory.

For comparison, let's take a look at the bundle structure of the IOUSBFamily.kext shown in Figure 6. The directory structure should look familiar. The Contents directoy contains the Info.plist, Resources, and the MacOS directories as you'd expect. But, the IOUSBFamily.kext has a Plugins directory full of other KEXTs. The Plugins directory provides a mechanism for organizing a suite of drivers into a single bundle. The I/O Kit will look one level deep into a KEXT for additional KEXTs located in the Plugins folder. All of the KEXTs will be available for driver matching as if they were organized individually.

Figure 6. Directory Structure of IOUSBFamily.kext



Property Lists

The Info.plist property list is the most important and the only required part of a KEXT bundle. The Info.plist contains essential configuration information and matching dictionaries. This information is readily available to system and program code at runtime. If viewed from a text editor, the property list would be in XML (Extended Markup Language) format. However, the Info.plist file is normally edited within Project Builder.

The Info.plist, information property list, is a collection of key-value pairs where the XML tags <key> and </key> enclose the key. Immediately following the key are the tags enclosing the value; these tags indicate the data type of the value; for example,

```
<key>idProduct</key> integer>10256</integer>
```

would define an integer value for the idProduct key.

Code Listing 1, shows the contents of the Cypress USB Storage driver Info.plist in XML format and Figure 7, shows the Info.plist in Project Builder.

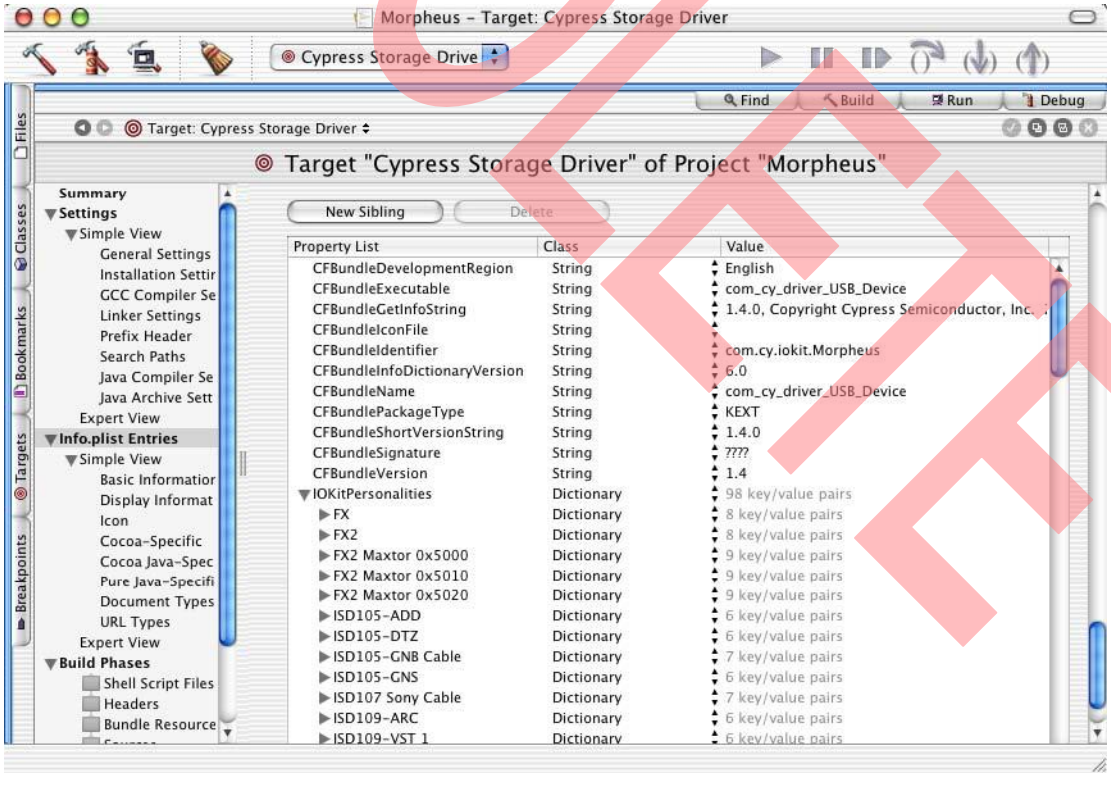
Code Listing 1 – Info.plist for the Cypress USB Storage driver.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
3 <plist version="1.0">
4 <dict>
5       <key>CFBundleDevelopmentRegion</key>
6       <string>English</string>
7       <key>CFBundleExecutable</key>
8       <string>com_cy_driver_USB_Device</string>
9       <key>CFBundleGetInfoString</key>
10      <string>1.4.0, Copyright Cypress Semiconductor, Inc. 2000-2002</string>
11      <key>CFBundleIdentifier</key>
12      <string>com.cy.iokit.Morpheus</string>
13      <key>CFBundleInfoDictionaryVersion</key>
14      <string>6.0</string>
15      <key>CFBundleName</key>
16      <string>com_cy_driver_USB_Device</string>
```

OBSOLETE

```
17      <key>CFBundlePackageType</key>
18      <string>KEXT</string>
19      <key>CFBundleShortVersionString</key>
20      <string>1.4.0</string>
21      <key>CFBundleSignature</key>
22      <string>????</string>
23      <key>CFBundleVersion</key>
24      <string>1.4</string>
25      <key>IOKitPersonalities</key>
26      <dict>
27
28  <!-- ... matching dictionaries not shown ... -->
29
30  </dict>
31      <key>OSBundleCompatibleVersion</key>
32      <string>1.0.0</string>
33      <key>OSBundleLibraries</key>
34      <dict>
35          <key>com.apple.iokit.IOSCSIArchitectureModelFamily</key>
36          <string>1.0.0</string>
37          <key>com.apple.iokit.IOStorageFamily</key>
38          <string>1.1</string>
39          <key>com.apple.iokit.IOUSBFamily</key>
40          <string>1.8</string>
41      </dict>
42      <key>OSBundleRequired</key>
43      <string>Local-Root</string>
44  </dict>
45  </plist>
46
```

Figure 7. Info.plist in Project Builder



OBSOLETE

Summary of Property List Keys.

CFBundleDevelopmentRegion – defined in IOCFBundle.h. This key identifies the region code the property list strings are written in.

CFBundleIdentifier – defined in IOCFBundle.h. This key is used to uniquely identify the bundle in the system. Use the reverse DNS of your company (e.g., “com.yourCo.driver”).

CFBundleInfoDictionaryVersion – defined in IOCFBundle.h. The version number for the Info.plist format. New plists should be version 6.0.

CFBundleVersion – defined in IOCFBundle.h. This key is the bundle version string written in Mac OS 9 vers resource style (e.g., “1.2.3b4”).

CFBundleName – defined in IOCFBundle.h. The short display name for the bundle, this key is more applicable for applications.

CFBundleExecutable – defined in IOCFBundle.h. The name of the file holding the executable code for the bundle.

CFBundlePackageType – The MacOS-style four-letter-type that identifies that this is a KEXT bundle.

CFBundleSignature –The MacOS-style four-letter-type that identifies the bundle creator. This is the application creator type. Not relevant for KEXTs.

CFBundleShortVersionString – This key is new with Mac OS X 10.2. This is a short version string (e.g.,“1.2.3”)

CFBundleGetInfoString – The string for Get Info panels in applications. May be used in upcoming versions of Apple System Profiler for KEXTs.

IOBundleLibraries – These are your KEXT’s dependencies. Here you list the other KEXTs that your KEXT needs in order to load and run properly.

OSBundleRequired – This tells IOKit if this driver is needed for early, boot time, driver matching. Set this to Local-Roott if you need to load your KEXT at boot time. However, if you need to compete with OS provided drivers like USB Mass Storage Class or USB HID Class drivers, which are loaded early in the boot processes, set this to Root.

IOKitPersonalities – this is a dictionary containing one or more matching dictionaries for this KEXT. Each matching dictionary defines a different personality for the KEXT. It is possible for a driver with multiple personalities to be instantiated more than once if several personalities match. Your driver can have more than one personality for a variety of reasons. It could be that the driver (as packaged in the KEXT) supports more than one type of device, or multiple versions of the same type of device, or you have multiple drivers packaged in the KEXT.

IOKitPersonalities dictionaries have a set of key-value pairs that are used by I/O Kit for driver matching. Some are common to all personalities (like CFBundleIdentifier, IOClass, and IOProviderClass), others are defined by a family. The IOUSB-Family defines the keys found in IOKitPersonalities dictionaries for USB drivers. IOUSBFamily passive matching criteria is from the USB Common Class Specification section 3.10. Add the fields from the specification (e.g., idProduct, and idDevice) to the personality.Warning: Follow the specification to the letter. If you add one extra key-value pair more than specified, the matching dictionary will fail!

CFBundleIdentifier – a matching dictionary key. This identifies the KEXT containing the IOClass that will be loaded. Normally, the CFBundleIdentifier and IOClass point to this KEXT, but it’s not required. It is perfectly valid for a KEXT to have a matching dictionary that matches and launches to a driver in another KEXT.

IOClass – a matching dictionary key. The name of the C++ driver class I/O Kit will instantiate for probing.

IOProviderClass – a matching dictionary key. This key identifies the name of the nub class that this personality matches to. For USB drivers the IOProviderClass will be IOUSBDevice or IOUSBInterface.

IOKitDebug – an optional key used to turn on I/O Kit debugging. This key makes I/O Kit dump additional data to the system.log. This can be helpful when debugging drivers. This is especially helpful when you are having difficulty getting your driver to load (most likely a matching problem).

Code Listing 2 – Info.plist for the Cypress USB Storage driver.

```
1 <key>IOKitPersonalities</key>
2      <dict>
3 <!-- Each personality has a different name. -->
4          <key>FX</key>
5          <dict>
6 <!-- The name of the bundle the IOClass will be called from. Usually, this will be the same
7 -- as the CFBundleIdentifier for this KEXT but it doesn’t have to be. -->
8             <key>CFBundleIdentifier</key>
9             <string>com.cy.iokit.Morpheus</string>
10
11 <!-- The name of the class IOKit will instantiate when probing. Notice this class is
different
12 -- for each of three example matching dictionaries.
13 -->
14            <key>IOClass</key>
15            <string>com_isd_driver_CYMSC_Device</string>
16
```

OBSOLETE

```
<!-- IOKit matching properties
-- All drivers must include the IOProviderClass key, giving
-- the name of the nub class that they attach to. The provider
-- class then determines the remaining match keys. A personality
-- matches if all match keys do; it is possible for a driver
-- with multiple personalities to be instantiated more than once
-- if several personalities match.
-->
          <key>IOProviderClass</key>
          <string>IOUSBInterface</string>

<!-- IOUSBFamily passive matching criteria.
-- This personality matches on IOUSBInterface so the following key-value pairs identify that
-- interface.
-->
          <key>bConfigurationValue</key>
          <integer>1</integer>
          <key>bInterfaceNumber</key>
          <integer>0</integer>
          <key>idProduct</key>
          <integer>10256</integer>
          <key>idVendor</key>
          <integer>1351</integer>
       </dict>

       <key>ISD105-GNS</key>
       <dict>
          <key>CFBundleIdentifier</key>
          <string>com.cy.iokit.Morpheus</string>
          <key>IOClass</key>
          <string>com_isd_driver_USS725_Device</string>

<!-- IOKit matching properties
-- This personality will match on an IOUSBDevice object.
-->
          <key>IOProviderClass</key>
          <string>IOUSBDevice</string>

<!-- IOUSBFamily passive matching criteria.
-- idProduct and idVendor identify this driver as a USB device driver.
-->
          <key>idProduct</key>
          <integer>513</integer>
          <key>idVendor</key>
          <integer>1451</integer>
       </dict>

       <key>ISD200-MSC ATAPI</key>
       <dict>
          <key>CFBundleIdentifier</key>
          <string>com.cy.iokit.Morpheus</string>
          <key>IOClass</key>
          <string>com_isd_driver_ISDMSC_Device</string>
          <key>IOKitDebug</key>
          <integer>65535</integer>
          <key>IOProviderClass</key>
          <string>IOUSBDevice</string>
          <key>idProduct</key>
          <integer>48</integer>
          <key>idVendor</key>
          <integer>1451</integer>
       </dict>
<!-- ... some dictionaries not shown ... -->
    </dict>

```

OBSOLETE

IOService Startup Sequence – Driver Loading

The IOService object controls the startup sequence of the driver life cycle. Since all objects in I/O Kit inherit from IOService it's important to understand the steps that occur. During the startup sequence there are two tasks that occur – driver matching and driver startup. A developer can override any of these methods, so is important to know the order methods are called and their function.

Driver matching

- init – called by I/O Kit when KMOD is loaded into memory.
- attach – called by the provider to temporarily attach the KEXT to it in the I/O Registry.
- probe – called by I/O Kit to determine if KEXT supports the device.
- detach – called after the probe method, regardless if the probe passed or failed. The driver needs to be aware that this will occur.
- free – called if probe fails.

Driver startup

- attach – called if you pass the probe method. The KEXT is put into the I/O Registry.
- start – called to do one time initialization and perform I/O with the device to complete the matching process.

What happens when a USB Device is removed?

When a device is removed from the bus, the hub driver notices that the device has been removed and it issues a terminate message to the IOUSBDevice object. The IOUSBDevice object then sends the terminate message to any of its child objects, which in turn send messages to their children and so forth until the leaf node of the tree is notified that the device has gone away. The leaf node driver must close its connection to its parent, terminate its state machine, and prepare to be removed from memory. Again, the IOService object controls this part of the driver life cycle.

IOService Termination Sequence

Tearing down a driver stack is a little bit more complicated then building it up. A driver stack must be released in a coordinated manner. Drivers must stop accepting new requests and clear out all queued and in-progress work.

The I/O Kit performs an orderly tear-down of a driver stack upon device removal in three phases.

Phase 1 – Make the driver objects inactive.

The first phase involves making the driver stack inactive so that it receives no new I/O requests. The driver's terminate method will be called by the hub driver. The default behavior of the terminate message is to make the object inactive immediately. As a consequence of being made inactive, each object also sends its clients a kIOServicesIsTerminated message via the message method. [The kIOServicesIsTerminated message has been deprecated. If a kernel driver receives this message it should no longer be handled.]

Phase 2 – Clear pending and in-progress I/O requests from driver queues.

I/O Kit calls the willTerminate method and the driver needs to cancel or abort all pending I/O. The driver can cancel the I/O asynchronously – it doesn't have to wait for the call to return.

Then I/O Kit will call the didTerminate method. At this point termination is almost complete. The driver needs to wait for all outstanding I/O to complete. Once all I/O is done, the driver needs to close its provider (i.e., call the close method on the provider object).

Phase 3 – Clean up.

Finally, in the third phase, the I/O Kit invokes the appropriate driver life-cycle methods on drivers to clean up allocated resources and detach the objects from the I/O Registry before freeing them. The stop method will be called and the driver needs to release or free any resources allocated in the start method. I/O Kit then calls the detach method in the driver – the detach method removes the driver from the I/O Registry.

IOUSBFamily Workloop

The workloop is one of the most misunderstood technologies to kernel driver developers. Every driver has to work on the workloop. There is one workloop for every interrupt source (one per IOUSBController – most Macs have two USB controllers). Every member of the USBFamily that is attached to a particular controller and every driver for a USB device attached to that USB controller, participates on the same work-loop. The workloop is a serialization mechanism for I/O calls. What this means to developers is that your driver has to be careful about kind of calls it makes.

For example, a synchronous call to USB from an asynchronous callback routine will deadlock the system. In order for the synchronous call to complete, an interrupt has to come in on a different thread but that interrupt is not able to execute on the workloop because the workloop is locked by your driver's callback routine.Thus, a deadlock occurs. A kernel driver developer needs to be very aware of the execution context when developing drivers.

There are two mechanisms that are useful in working with workloop execution contexts. The getWorkloop method returns the USB controller's Workloop. This is useful when you want to do work on the Workloop. And if you want to do work outside of the Workloop you can use the thread_call mechanism.

Writing a driver using I/O Kit Inheritance

Driver inheritance in I/O Kit has confused a lot of developers being exposed to I/O Kit for the first time. At first it may seem logical that a USB driver would inherit from classes in the IOUSBFamily, but this is not the case. USB drivers are not members of and do not inherit from the IOUSBFamily. They use the IOUSBFamily for their transport mechanism and are thus clients of the USB family classes. Let's look at some examples.

The AppleUSBAudioDevice class is a subclass of the IOAudioDevice which is a subclass of IOService. IOService being the base class for all I/O Kit objects. So this driver is a member of the Audio Family.

OBSOLETE

```
Client
AppleUSBAudioDevice <- IOAudioDevice <-
IOService
IOUSBInterface (provider)
```

The IOUSBHIDDriver class is a subclass of the IOHIDDevice which is a subclass of IOService. IOService being the base class for all I/O Kit objects. So this driver is a member of the HID Family. The fact that this driver is bundled with the IOUSBFamily only serves to confuse us.

```
IOHIDSystem (client)
IOUSBHIDDriver <-IOHIDDevice <- IOService
IOUSBInterface (provider)
```

Code-less Kernel Extension Examples

Code-less kernel extensions are kernel extensions that have no code, only an Info.plist. A code-less kernel extension will provide a personality that I/O Kit will match to the USB device but the personality will point to another kernel extention. The following examples are not only nifty tricks for solving the problems presented, but they also provide additional insight in to the way driver matching works on Mac OS X.

Classic Environment HID Example

Problem: Suppose you have a USB HID device that you want to make available only to the Classic environment.

Solution: Create a code-less KEXT that will match to the device instead of Apple's IOUSBHIDDriver and prevent the IOUSBHIDDriver from taking control of the device. This requires two personalities and the default behavior of the IOService object and the AppleUSBMergeNub driver. Code Listing 3 shows what the Info.plist would look like.

Code Listing 3 – Classic Enviornment HID

```
<key>IOKitPersonalities</key>
<dict>
    <key>Merge driver</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBMer
geNub</string>
        <key>IOClass</key>
        <string>AppleUSBMergeNub</string>
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idProduct</key>
        <integer>10256</integer>
        <key>idVendor</key>
        <integer>1351</integer>
        <key>IOProviderMergeProperties</key>
        <dict>
            <key>ClassicMustSeize</key>
            < true/>
        </dict>
<key>IOService driver</key>
<dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.kerneliokit</string>
        <key>IOClass</key>
        <string>IOService</string>
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idProduct</key>
        <integer>10256</integer>
    <key>idVendor</key>
    <integer>1351</integer>
    </dict>
    </dict>
</dict>
<key>OSBundleLibraries</key>
<dict>
    <key>com.apple.iokit.IOUSBFamily</key>
    <string>1.8.2</string>
</dict>
<key>OSBundleRequired</key>
<string>Local-Root</string>
```

Both the Merge driver and the IOService driver personalities match the device so both will be probed. When the Merge driver personality matches, the AppleUSBMergeNub driver's probe method is called. The AppleUSBMergeNub driver copies the contents of its IOProviderMergeProperties dictionary to its provider's dictionary. This puts the ClassicMustSeize key-value pair entry into its provider's dictionary in the I/O Registry.

The provider in this case is the IOUSBDevice nub. After the contents of the IOProviderMergeProperties dictionary has been copied, the AppleUSBMergeNub fails the probe method – telling I/O Kit that it doesn't support this device.

The IOService object's personality will match. Then the IOService object's start method will be called which simply returns true. So we get a driver that returns true from the start method but doesn't actually do anything with the device. Now, because of the ClassicMustSeize key-value pair entry in the I/O Registry, Classic drivers are able to open the HID device and take control of it.

Also, the property list must include the OSBundleRequired entry so that this KEXT can compete with the Apple supplied HID driver at boot time.

Vendor-Specific Composite Device Example

Problem: Suppose you have a vendor-specific device whose functionality is implemented in the device's interfaces. We want to have the interfaces of the device created so then other class drivers or vendor-specific interface drivers can match to the interfaces.

Solution: Create a code-less KEXT with a personality that matches to the device but points to the AppleUSBComposite driver. The AppleUSBComposite driver's main jobs are to set the configuration on the device, which creates the interfaces, and to handle any reconfiguration tasks after resets occur. The contents of the Info.plist are shown in Code Listing 4.

Code Listing 4– Vendor-Specific Composite Device

```
<key>IOKitPersonalities</key>
<dict>
    <key>V-S Composite driver</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSB-
    Composite</string>
        <key>IOClass</key>
        <string>AppleUSBComposite</string>
```

OBSOLETE

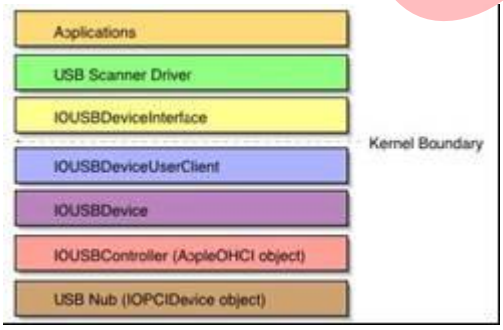
```
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idProduct</key>
        <integer>10256</integer>
        <key>idVendor</key>
        <integer>1351</integer>
    </dict>
</dict>
```

USB User Space Driver Development

User space is where most USB drivers will reside. User-space drivers are easier to develop and easier to debug than kernel drivers. One can use CodeWarrior's or Project Builder's source level debugging tools and crashes in a user-space driver don't bring down the system.

The device interface mechanism provided by I/O Kit provides user-space drivers and applications with a means of communicating with hardware from outside the kernel. The IOUSB-Family provides two types of device interfaces: IOUSBDeviceInterface for communicating with the device itself and IOUSBInterfaceInterface for communicating with an interface on the device. Figure 8, shows the driver stack with a user space USB scanner driver communicating with an IOUSBDeviceInterface. Notice the kernel boundary is bracketed with a IOUSBDeviceInterface above the kernel and a IOUSBDeviceUserClient in the kernel.

Figure 8. User Space Driver Stack



The UserClient and the DeviceInterface objects always come in pairs to bridge the kernel boundary. The IOUSBFamily provides two UserClient classes: IOUSBInterfaceUserClient and IOUSBDeviceUserCient. These two classes allow user-space drivers to communicate with the three main kernel objects: IOUSBPipe, IOUSBInterface and IOUSBDevice.

IOUSBFamily Kernel Classes

IOUSBPipe

IOUSBInterface <--> IOUSBInterfaceUserClient

IOUSBDevice <--> IOUSBDeviceUserClient

When the IOUSBDevice object is instantiated, it creates the IOUSBDeviceUserClient kernel object. If a kernel-space driver calls the open method on IOUSBDevice, it obtains exclusive access to the device. When the open method is called, the IOUSBDevice removes the IOUSBDeviceUserClient from the driver stack. This prevents user-space drivers from being able to access the device.

If a kernel-space driver has not opened the IOUSBDevice, the IOUSBDeviceUserClient is available to user-space drivers. A user-space driver uses a IOUSBDeviceInterface object to find and access the IOUSBDeviceUserClient object. When the user-space driver calls the USBDeviceOpen method, it obtains exclusive access to the device preventing other user-space drivers from accessing it.

User-space drivers do not compete in the matching process at the same time as kernel-space drivers. In some cases a code-less kernel extension may have to be used to claim the device. Remember that you have three environments, Classic, kernel space and user space, that may be vying for a shot at the device.

Using Device Interfaces

The procedure for finding and using a device interfaces is long but very consistent. The steps are outlined here.

1. Obtain Master Port – the first step is to create a mach port to access the kernel.

```
#include <mach/mach.h>
err = IOMasterPort(MACH_PORT, NULL,
&masterPort);
```

The masterPort should be released when done.

```
mach_port_deallocate(…)
```

2. Create a matching dictionary for the device type we want to find.

For IOUSBDevice objects:

```
MathingDictionary = IOServiceMatching(
kIOUSBDevice-ClassName )
```

For IOUSBInterface objects:

```
MatchingDictionary = IOServiceMatching(
kIOSUBIterfaceClassName )
```

3. Add qualifications to the matching dictionary – This is a family specific step. For USB devices we need to use the USB Common Class Specification section 3.10 as our guide. To match the second line of the chart in section 3.10 of the USB Common Class Specification, you must include the two items (idProduct and idVendor). Do not include any other fields in your matching dictionary device class or device subclass or your matching dictionary will fail. The matching rules are very strict!

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

OBSOLETE

```
CFNumberCreate( kCFAllocatorDefault,
kCFNumberSInt32Type, &usbProduct));
```

4. Obtain the kernel iterator.

```
err = IOServiceGetMatchingServices(
masterPort, matching-Dictionary, &iterator
);
while ( usbDeviceRef =
IOIteratorNext(iterator))
```

The iterator should be released when done.

```
IOObjectRelease(iterator);
```

5. Instantiate the device interface. This is the call that instantiates the UserClient and DeviceInterface pair of objects and sets up the communication channel which allows user space code to access the kernel objects.

```
IOCreatePlugInterfaceForService(…)
```

The object returned does not yet represent the interface you want. You must first call QueryInterface to say that you are looking for this particular type of interface.

```
(*iodev)->QueryInterface(…, &dev);
(*iodev)->Release(iodev);
```

6. Initialize the IOUSBDeviceInterface and create the interfaces.

```
// First you open the device
(*dev)->USBDeviceOpen(dev)

// Get the configuration descriptor from the
device.
(*dev)->GetConfigurationDescriptorPtr(dev…)

// Set the configuration to be used. This
will create the interfaces.
(*dev)->SetConfiguration(dev,…)

// Create interface iterator tokens for the
various IOUSBInterfaceInterfaces.
(*dev)->CreateInterfaceIterator(…)

// Close and release the device. You'll be
using the interfaces from here on.
(*dev)->USBDeviceClose(dev)
(*dev)->Release(dev)
```

7. Using IOUSBInteraceInterface to transfer data.

```
// Open exclusive access to the interface.
(*intf)->USBInterfaceOpen(intf)

// Select an alternate interface in this
configuration.
(*intf)->SetAlternateInterface(intf,…)

// Determine the number of endpoints in this
interface.
(*intf)->GetNumEndpoints(intf…)

// Use the GetPipeProperties to identify the
pipes (Bulk-In, Bulk-Out, Isoc-In, etc.)
(*intf)->GetPipeProperties(intf,…)

// – data transfer, yeah!
(*intf)->ReadPipe(intf,…)
(*intf)->WritePipe(intf,…)

// clean-up when the driver is finished
(*intf)->USBInterfaceClose(intf)
(*intf)->Release(intf)
```

A Note on Using Composite Devices from User Space

If the device is a composite class device with no vendor-specific driver to match against it, the AppleUSBComposite driver matches against it and starts up as its provider. The AppleUSBComposite driver then configures the device by setting the first configuration in the device's list of configuration descriptors. This causes the IOUSBFamily to abstract each interface descriptor in the chosen configuration into IOUSBInterface nub objects. These nub objects are attached to the I/O Registry as children of the original IOUSBDevice nub object and are registered for matching with the I/O Kit. Because the AppleUSBComposite driver configured the device, setting the configuration again from your application will result in the destruction of the IOUSBInterface nub objects and the creation of new ones. The only reason to set the configuration of a composite class device that's matched by the AppleUS-BComposite driver is to choose a configuration other than the first one. For other types of devices, non-composite class devices or composite class devices with vendor-specific drivers that match against them, there is no guarantee that any configuration will be set and you may have to perform this task within your application or user-space drivers.

User Space Drive Notification

User-space drivers can register with I/O Kit's notification mechanisms to be notified about device appearance, device removal, changes in the state of the device, changes in system power and a broad range of other events.

Sharing Devices – The Device Sharing Model

The I/O Kit model and the IOUSBFamily force exclusive access to devices. However, there is a protocol for user-space and kernel-space drivers to communicate a desire to share a device. Drivers need to add support for the following messages in order to play nice together.

- kIOMessageServiceIsRequestingClose – is received when another entity is requesting access to a device.
- kIOMessageServiceIsAttemptingOpen – is received when another entity has attempted to open a device. It's a clue that someone else would like to use the device and a driver should close down access to the device if possible.

  A driver can also register for I/O Kit notification when a driver has closed its access to the device. The notification set to the driver will be the message:

OBSOLETE

- kIOMessageServiceWasClosed

Using Synchronous and Asynchronous calls from User Space

Blocking I/O for User Space drivers

There are many USB APIs that can "block" until the USB transaction completes. Using blocking I/O can simplify driver designs. Synchronous calls from user-space drivers are something that is always safe to do. You make a synchronous call, your thread blocks, and when the call completes, your thread picks up again.

Graphical user interface applications that directly access USB devices should create a seperate thread for controlling the USB device from the thread(s) controlling the user interface. Otherwise, the user interface will feel sluggish or may appear to the user to hang.

Asynchronous I/O in User Space

In user space you don't really have to be worried about the issues with synchronous versus asynchronous I/O, as is the case with kernel drivers. Because user space threads are never running on the workloop, deadlocks are not possible. However, if your driver design lends it's self better to doing asynchronous I/O, callbacks to user space are possible.

Callbacks to user space are a little more difficult as there is no mechanism in Mac OS X or I/O Kit to make direct calls from kernel threads to user threads. So, what happens is that a mach message is posted on a port from the kernel side and there is a user thread that will check that port using the CFRunLoop technology. When a callback message is found, the user thread dispatches the callback routine for the kernel thread.

In your user-space driver you need to create an asynchronous event source for your interface (IOUSBDeviceInterface or IOUSBInterfaceInterface) object. Then add that event source to your CFRunLoop (Documentation for CFRunLoop is in CFRunLoop.h.) Call CFRunLoopRun and your event source will get processed, calling back the callback routines you set up for your asynchronous calls.

```
(*intf)->CreateInterfaceAsyncEventSource(…)
CFRunLoopSource(…)
(*intf)->ReadPipeAsync(intf,…)
CFRunLoopRun();
```

USB in Classic

The Classic environment in Mac OS X uses the Mac OS 9 driver model – the entire Mac OS 9 USB stack is in Classic. Thus Mac OS 9 drivers should see no differences between running natively on Mac OS 9 or in the Classic environment.

The USB stack in the classic environment always attempts to capture two kinds of devices:

- Printing class devices, and
- Vendor specific class devices

Other devices are expected to have Mac OS X native support. USB device driver developers need to use the ClassicMust-Seize property or the Device Sharing Model to accommodate the interactions of classic, user-space and kernel-space drivers.

Where to Go Next

Everyone should start out reading the *System Overview* book found at /System/Documentation/Developer/SystemOverview/SystemOverview.pdf.

Kernel Information

General Darwin documentation can be found at http://developer.apple.com/techpubs/macosx/Darwin/kernel.html.

Read the *Kernel Environment* book found at /System/Documentation/Developer/Kernel/KernelEnvironment.pdf.

The book *IO Kit fundamentals* can be found at http://developer.apple.com/techpubs/macosx/Darwin/IOKit/IOKitFundamentals.

This book describes the features and architecture of the I/O Kit and discusses important concepts and mechanisms, including the I/O Registry, driver matching and loading, the class hierarchy, event handling, data management, and power management.

The book *Writing an I/O Kit Device Driver* can be found at

http://developer.apple.com/techpubs/macosx/Darwin/IOKit/DeviceDrivers/WritingKitDrivers/index.html.

This book discusses many of the issues related to device-driver development with the I/O Kit, including driver matching, I/O transactions, programming conventions, and debugging.

User Space Information

Accessing Hardware From Applications found at http://developer.apple.com/techpubs/macosx/Darwin/IOKit/DeviceInterfaces/AccessingHardware/index.html.

This book describes how applications and other user programs, using the APIs of the I/O Kit framework, can access hardware by communicating with the kernel. It explains what device interfaces are and how to use them. And it describes how to locate the device files of certain devices that can then be accessed with POSIX APIs.

Also read *Working With USB Device Interfaces* found at http://developer.apple.com/techpubs/macosx/Darwin/IOKit/DeviceInterfaces/USBBook/index.html.

The USB Technology Home Page can be found at http://developer.apple.com/hardware/usb/index.html.

PROFILE

More USB Documentation

Technical Q&As

- Tips on USB driver matching for Mac OS X

http://developer.apple.com/qa/qa2001/qa1076.html

- Making sense of IO Kit error codes

http://developer.apple.com/qa/qa2001/qa1075.html

- Issues with boot time KEXT loading

http://developer.apple.com/qa/qa2001/qa1087.html

Everyone should start out reading the *System Overview* book found at

/System/Documentation/Developer/SystemOverview/SystemOverview.pdf.

Document History

Document Title: Mac OS X: Getting Started with USB – AN1105

Document Number: 001-65423

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3095309	NXZ	11/25/2010	AN1105 spec updated to new application note template.
*A	4108854	RSKV	08/30/2013	Obsolete application note

OBSOLETE

All product and company names mentioned in this document are the trademarks of their respective holders.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/

© Cypress Semiconductor Corporation, 2002-2013. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.