Driver and Device Matching

Before a device—or any service provider—can be used, a driver for it must be found and loaded into the kernel. The I/O Kit defines a flexible, three-phase matching process that narrows a pool of candidate drivers down to one or more drivers. The final candidate (or, if multiple candidates, the most eligible one) is then loaded and given the first opportunity to manage the device or service provider.

The matching process makes use of the matching dictionaries defined as XML key-value pairs in a driver's information property list. Each matching dictionary specifies a personality of the driver, which declares its suitability for a device or service of a particular type.

This chapter discusses driver personalities and the matching language that describes them. It then describes the matching process, which uses the information in the driver personalities to identify the most appropriate driver for a detected device. The chapter also briefly discusses the device-matching procedure that applications use for loading device interfaces. See *Accessing Hardware From Applications* for the complete details.

Driver Personalities and Matching Languages

Each device driver, considered as a loadable kernel extension (KEXT), must define one or more personalities that specify the kinds of devices it can support. This information is stored in XML matching dictionaries defined in the information property list (`Info.plist`) in the driver's KEXT bundle. A dictionary in this sense 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,

```
<integer>74562</integer>
```

would define an integer value.

Each matching dictionary is itself contained within the information property list's `IOKitPersonalities` dictionary.

The dictionary values of a personality specify whether a driver is a candidate for a particular device. All values in the personality must match for the driver to be selected for the device; in other words, a logical AND is performed on the values. Some of the keys may take a list of space-delimited values, which are generally examined in an OR fashion. Thus you might have a "model" key for a certain PCI card driver personality that takes a list of model numbers, each identifying a supported model from a specific card vendor.

The specific keys that are required depend on the family. A driver for a PCI card, for example, can define a value that is checked against the PCI vendor and device ID registers. Some families, such as the PCI family, provide fairly elaborate matching strategies. For instance, consider this key-value pair:

```
<key>IOPCIMatch</key>
<string>0x00789004&0x00ffffff 0x78009004&0xff00ffff</string>
```

This expression, which is used to match various Adaptec SCSI cards, consists of two compound values, each of which can be a valid match. To evaluate these values, the driver family reads the 32-bit vendor and device ID from the PCI card and masks it with the value to the right of each ampersand. The result of that operation is then compared with the value to the left of the ampersand to determine if there is a match.

Listing 4-1 shows a partial listing of a driver personality from the XML file for an Ethernet controller driver.

Listing 4-1 A partial listing of an XML personality for an Ethernet controller

```
<key>IOKitPersonalities</key>
   <dict>
       <dict>
           <!-- Each personality has a different name. -->
           <key>Name</key>      <string>PCI Matching</string>

           <!-- ... some keys not shown ... -->

           <!-- The name of the class IOKit will instantiate when probing. -->
           <key>IOClass</key>   <string>ExampleIntel82558</string>

           <!-- 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>IOPCIDevice</string>

           <!-- IOPCIDevice matching uses any of four possible PCI match
             -- criteria. This personality just uses IOPCIMatch to check the
             -- device/vendor ID.
             -->
           <key>IOPCIMatch</key>
               <string>0x12298086</string>

           <!-- The initial match score for this personality.-->
           <key>IOProbeScore</key>    <integer>400</integer>
       </dict>

       <dict>
           <!-- Can have additional personalities. -->
           <!-- ... (not shown) -->
       </dict>
   </dict>
```

As mentioned in Listing 4-1 every driver must include the `IOProviderClass` key with a value that identifies the nub to which the driver attaches. In very rare cases, a driver might declare `IOResources` as the value of its `IOProviderClass` key. `IOResources` is a special nub attached to the root of the I/O Registry that makes resources, such as the BSD kernel, available throughout the system. Traditionally, drivers of virtual devices match on `IOResources` because virtual devices do not publish nubs of their

own. Another example of such a driver is the HelloIOKit KEXT (described in Creating a Device Driver with Xcode) which matches on `IOResources` because it does not control any hardware.

> **Important:** Any driver that declares `IOResources` as the value of its `IOProviderClass` key must also include in its personality the `IOMatchCategory` key and a private match category value. This prevents the driver from matching exclusively on the `IOResources` nub and thereby preventing other drivers from matching on it. It also prevents the driver from having to compete with all other drivers that need to match on `IOResources`. The value of the `IOMatchCategory` property should be identical to the value of the driver's `IOClass` property, which is the driver's class name in reverse-DNS notation with underbars instead of dots, such as `com_MyCompany_driver_MyDriver`.

Because a driver can contain multiple matching dictionaries, each one defining a different personality for the driver, the same driver code can be loaded for different devices. For purposes of competition, the I/O Kit treats each personality as if it were a driver. If, in any single personality, all of the properties required by the family match, the driver's code is loaded and given a chance to run for that device.

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 more commonly, multiple versions of the same type of device. Another reason might be that the driver supports similar devices, each of which is attached to the system on different buses; for example, Zip drives can be attached to USB, FireWire, SCSI, ATAPI, and other buses. Because each of these attaches to a different nub class, it has different matching values. The personalities of a driver can also range from device-generic to device-specific. The personalities of the AppleUSBAudio driver (Listing 4-2) illustrate this.

Listing 4-2 Driver personalities for the AppleUSBAudio driver

```
<key>IOKitPersonalities</key>
<dict>
    <key>AppleUSBAudioControl</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBAudio</string>
        <key>IOClass</key>
        <string>AppleUSBAudioDevice</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>bInterfaceClass</key>
        <integer>1</integer>
        <key>bInterfaceSubClass</key>
        <integer>1</integer>
    </dict>
    <key>AppleUSBAudioStream</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.apple.driver.AppleUSBAudio</string>
        <key>IOClass</key>
        <string>AppleUSBAudioDMAEngine</string>
        <key>IOProviderClass</key>
        <string>IOUSBInterface</string>
        <key>bInterfaceClass</key>
```

```
            <integer>1</integer>
            <key>bInterfaceSubClass</key>
            <integer>2</integer>
        </dict>
        <key>AppleUSBTrinityAudioControl</key>
        <dict>
            <key>CFBundleIdentifier</key>
            <string>com.apple.driver.AppleUSBAudio</string>
            <key>IOClass</key>
            <string>AppleUSBTrinityAudioDevice</string>
            <key>IOProviderClass</key>
            <string>IOUSBInterface</string>
            <key>bConfigurationValue</key>
            <integer>1</integer>
            <key>bInterfaceNumber</key>
            <integer>0</integer>
            <key>idProduct</key>
            <integer>4353</integer>
            <key>idVendor</key>
            <integer>1452</integer>
        </dict>
    </dict>
```

This matching dictionary defines three personalities: `AppleUSBAudioControl`, `AppleUSBAudioStream`, and `AppleUSBTrinityAudioControl`. In matching for a detected USB Trinity audio-control device, the `AppleUSBTrinityAudioControl` would be chosen; for any other audio-control device, the generic personality (`AppleUSBAudioControl`) would match.

One common property of personalities is the probe score. A probe score is an integer that reflects how well-suited a driver is to drive a particular device. A driver may have an initial probe-score value in its personality and it may implement a `probe` function that allows it to modify this default value, based on its suitability to drive a device. As with other matching values, probe scores are specific to each family. That's because once matching proceeds past the class-matching stage, only personalities from the same family compete. For more information on probe scores and what a driver does in the `probe` function, see Device Probing

Driver Matching and Loading

At boot time and at any time devices are added or removed, the process of driver matching occurs for each detected device (or other service provider). The process dynamically locates the most suitable driver in `/System/Library/Extensions` for the device or service.

As described in Driver Matching in the chapter Architectural Overview the matching process is triggered when a bus controller driver scans its bus and detects a new device attached to it. For each detected device the controller driver creates a nub. The I/O Kit then initiates the matching process and obtains the values from the device to use in matching (for example, examining the PCI registers). Once a suitable driver is found for the nub, the driver is registered and loaded. That driver, in turn, may create its own nub (possibly through behavior inherited from its family), which initiates the matching process to find a suitable driver.

Driver Matching

When a nub detects a device, the I/O Kit finds and loads a driver for the nub in three distinct phases, using a subtractive process. In each phase, drivers that are *not* considered to be likely candidates for a match are subtracted from the total pool of possible candidates until a successful candidate is found.

The matching process proceeds as follows:

1. In the **class matching** step, the I/O Kit narrows the list of potential drivers by eliminating any drivers of the wrong class for the *provider* service (that is, the nub). For example, all driver objects that descend from a SCSI class can be ruled out when the search is for a USB driver.
2. In the **passive matching** step, the driver's personality (specified in a driver's XML information property list) is examined for properties specific to the provider's family. For example, the personality might specify a particular vendor name.
3. In the **active matching** step, the driver's `probe` function is called with reference to the nub it is being matched against. This function allows the driver to communicate with the device and verify that it can in fact drive it. The driver returns a probe score that reflects its ability to drive the device. See Device Probing for more information. During active matching, the I/O Kit loads and probes all candidate drivers, then sorts them in order of highest to lowest probe score.

The I/O Kit then chooses the remaining driver with the highest probe score and starts it. If the driver successfully starts, it is added to the I/O Registry and any remaining driver candidates are discarded. If it does not start successfully, the driver with the next highest probe score is started, and so on. If more than one driver is in the pool of possible candidates, the more generic driver typically loses out to the more specific driver if both claim to be able to drive the device.

Device Probing

During the active matching phase, the I/O Kit requests each driver in the pool of remaining candidates to probe the device to determine if they can drive it. The I/O Kit calls a series of member functions defined in the IOService class and overridden in some cases by the driver's class. These functions, and the order in which they are called, are

```
init()
attach()
probe()
detach()
free() /* if probe fails */
```

These functions comprise the first part of a driver's life cycle (see Driver Object Life Cycle in the chapter The Base Classes for the full story). Note that four of these functions form complementary pairs, one nested inside the other: `init` and `free` are one pair, and `attach` and `detach` are the other.

During active matching, the code of a candidate driver is loaded and an instance of the principal class listed in the personality is created. The first function invoked is `init`, which is the `libkern` equivalent of the constructor function for the class. For I/O Kit drivers, this function takes as its sole argument an OSDictionary object containing the matching properties from the selected driver personality. The driver can use this to identify what specific personality it's been loaded for, determine what level of diagnostic output to produce, or otherwise establish basic operating parameters. I/O Kit drivers typically don't override the `init` function, performing their initialization in later stages.

However, if you do override the `init` function—or almost any other function of the driver life cycle—you must take care to do two things. The first is to invoke your superclass's implementation of the function. When you do this depends on the function; for example, in implementing `init` you should invoke the superclass's implementation as the first thing, and in `free` you should invoke it as the last statement of the function. The second general rule is that you should undo in the second function of a pair what you've done in the first function; thus, if you allocate memory for any reason in `init`, you should free that memory in `free`.

Next, the `attach` function (which is bracketed with the `detach` function) is called. The default implementation of `attach` attaches the driver to the nub through registration in the I/O Registry; the default implementation of `detach` detaches the driver from the nub. A driver can override the default implementations, but rarely needs to do so.

After `attach` the `probe` function is invoked. The I/O Kit always calls a driver's `probe` function if the driver's matching dictionary passively matches the provider (the nub). A driver may choose not to implement `probe`, in which case IOService's default implementation is invoked, which simply returns `this`.

The `probe` function takes as arguments the driver's provider and a pointer to a probe score. The probe score is a signed 32-bit integer initialized to a value specified in the driver's personality (or to zero if not explicitly initialized). The driver with the highest initial probe score is given the first chance to start operating the device. The purpose of the `probe` function is to offer drivers an opportunity to check the hardware and to modify their default probe scores as assigned in their personalities. A driver can check device-specific registers or attempt certain operations, adjusting its probe score up or down based on how well suited it is for the device it is examining. Whatever it finds, each driver must leave the hardware in the same state it was in when `probe` was invoked so the next driver can probe the hardware in its original state.

A driver, in its `probe` function, returns a driver object (`IOService *`) if the probe was successful and returns zero otherwise. The returned object is usually the driver itself, but the driver can return another driver that is more suited to the provider. The probe score is an in-out parameter, which `probe` can modify based on what it discovers about the device.

Driver Loading

After all drivers have probed the device, the one with the highest probe score is attached and its `start`function, which must be implemented by all drivers, is invoked. The `start` function initializes the device hardware and prepares it for operation. If the driver succeeds in starting, it returns `true`; the remaining candidate driver instances are discarded and the driver that started successfully continues operating. If the driver cannot initialize the hardware it must leave the hardware in the state it was in when `start` was invoked and return `false`. The failing driver is then detached and discarded, and the candidate driver with the next highest probe score is given a chance to start.

Some time after this occurs, all loaded drivers that are not currently in use are unloaded.

Device Matching

A user application that requires access to a device must first search for that device and then acquire the appropriate device interface to communicate with it. This process is known as device matching. Unlike driver matching, device matching searches the I/O Registry for a driver that is already loaded.

To perform device matching, follow these basic steps:

1. Establish a connection with the I/O Kit by obtaining a Mach port.
2. Define a dictionary that specifies the type of device to search for in the I/O Registry. The search can be refined by setting additional values in the dictionary. For example, a search for IOMedia objects can be narrowed down to find all ejectable media. You can find the values to match in the device header files (such as `IOSCSIDevice.h` or `IOATADevice.h`), by referring to the family-specific documentation, or by looking at the information property lists displayed in output from the I/O Registry Explorer application.
3. Obtain a list of all objects in the Registry that match your dictionary and choose the appropriate device.
4. Access the device you have chosen by obtaining a device interface for it. This step is explained more fully in Controlling Devices From Outside the Kernel

See the document*Accessing Hardware From Applications* for a full description of device matching.