Writing PCI Drivers

Hardware & Drivers > PCI and PC Card

2006-04-04

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

Simultaneously published in the United States and Canada.

Contents

Figures

About This Book

This document assumes some basic familiarity with programming the Mac OS X kernel. See *Kernel Programming Guide* for a broad overview.

Introduction to Writing PCI Drivers

Writing PCI Drivers is intended for anyone who wants to develop PCI drivers for Mac OS X. This book assumes a basic understanding of PCI (Peripheral Component Interface), as well as a basic understanding of the I/O Kit (Apple's object-oriented framework for developing device drivers in Mac OS X).

This book covers the issues specific to PCI driver development on Mac OS X. It does not cover the PCI architecture itself except as it pertains to the I/O Kit PCI framework. It also does not discuss general driver writing or porting. For books on these subjects, see "Other Apple Publications" (page 8).

This book only covers communication between your driver and PCI-based hardware. It does not cover the code you need to write to allow your driver to be accessed by the rest of the system. For information on writing a specific category of device driver, such as a network or video driver, see the documentation for the appropriate I/O Kit family.

Organization of this Document

Chapter 2, "PCI Family Architecture," (page 9) describes the basic I/O Kit classes in Mac OS X that are relevant if you are developing a PCI driver.

Chapter 3, "Writing a Driver for a PCI Bridge," (page 11) explains the basic types of PCI bridges and provides information on how to develop drivers for them.

Chapter 4, "Writing a Driver for a PCI Device," (page 15) covers matching and device setup for all PCI devices.

Chapter 5, "Writing a Driver for an AGP Device," (page 21) adds information specific to AGP devices.

Chapter 6, "Taking Primary Interrupts," (page 23) gives additional information about handling interrupts in a PCI device driver.

Chapter 7, "Endianness and Addressing," (page 29) explains the difference between byte-invariant addressing and register endianness and their importance to PCI driver developers.

See Also

Other Apple Publications

Apple has a series of documents on Mac OS X software development. You can obtain other books in this series from Apple's Developer Documentation website, http://developer.apple.com/Documentation.

Other documents that are of interest to device driver developers are *Kernel Programming Guide* and *I/O Kit Fundamentals*.

In addition, the book Designing PCI Cards and Drivers for Power Macintosh Computers, while not specific to Mac OS X, may be helpful in understanding PCI concepts and in understanding how Open Firmware and declaration ROMs interact with PCI devices. You can find this document in the hardware section of Apple's Developer Documentation website.

Information on the Web

Apple maintains several websites where developers can go for general and technical information on Mac OS X.

- The Darwin Documentation project website, http://www.opensource.apple.com/projects/documentation.
- Apple Product Information (http://www.apple.com/macosx)—provides general information on Mac OS X.
- Apple Developer Documentation (http://developer.apple.com/Documentation)—features the same documentation that is installed on Mac OS X, except that often the documentation is more up-to-date. Also includes legacy documentation.
- AppleCare Knowledge (http://www.apple.com/support/)—contains technical articles, tutorials, FAQs, technical notes, and other information.
- Apple Developer Connection Mac OS X Development page (http://developer.apple.com/macosx)—offers SDKs, release notes, product notes, product reviews, and other resources and information related to Mac OS X.

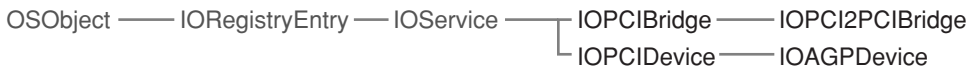
PCI Family Architecture

The PCI family defines the driver classes for PCI bridge controllers and the nub classes used by drivers of devices attached to a PCI bus. As a provider of PCI services, a PCI bridge controller driver scans the PCI bus and creates a nub for each device found. Each nub then starts the matching and loading process to find a driver for its PCI device, and the driver uses the nub to perform communication over the PCI bus.

Drivers for PCI bridge controllers are members of the PCI family, inheriting from a superclass within the family. Drivers of individual PCI devices are clients of the PCI family but are typically members of another family. A driver for a PCI Ethernet controller, for example, inherits from the Network family's `IOEthernetController` class but uses an instance of the PCI family's `IOPCIDevice` nub class to connect to the PCI bus. If you are writing a driver for a PCI device, you should read both this document and the document for the family from which your driver will actually inherit.

The PCI family is quite small, comprising only four classes. Figure 2-1 (page 9) shows the inheritance hierarchy for the PCI family. The two bridge classes, `IOPCIBridge` and `IOPCI2PCIBridge`, drive PCI host bridge controllers and PCI-to-PCI bridge controllers, respectively. `IOPCIBridge` is an abstract superclass that declares the general mechanism for a PCI bridge controller; hardware-specific subclasses implement this mechanism. `IOPCI2PCIBridge` is a concrete class that connects two hardware-specific bridge controller drivers.

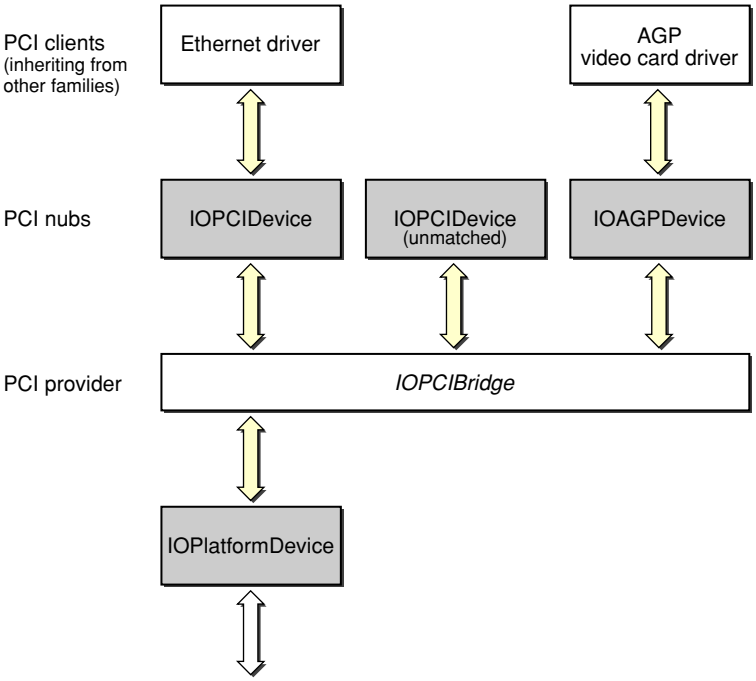
Figure 2-1 PCI family inheritance hierarchy



The two device classes, `IOPCIDevice` and `IOAGPDevice`, represent the access points for drivers of PCI and AGP devices. `IOPCIDevice` is the basic nub class for the PCI family, representing any PCI device in a PCI slot. For AGP devices, the nub class is `IOAGPDevice`. A driver object for a PCI device matches against a nub, using it to establish a connection to the PCI bus and to access the hardware registers on the device. Because `IOAGPDevice` is a subclass of `IOPCIDevice`, all of the `IOPCIDevice` methods can also be used on an `IOAGPDevice`.

Figure 2-2 (page 10) shows a typical arrangement of objects built on a PCI bus. Starting from the bottom, an `IOPlatformDevice` nub represents some controller built in to the logic board of the computer. In this case, it is a PCI host bridge controller. The appropriate instance of an `IOPCIBridge` subclass serves as the driver for the PCI bridge controller. The `IOPCIBridge` driver scans the PCI bus and creates an `IOPCIDevice` or `IOAGPDevice` for each device it finds; in this case it creates two regular PCI device nubs and an AGP device nub. These nubs then trigger driver matching and loading for their devices. One of the PCI devices turns out to be an Ethernet card, and the other an AGP video card. Because the remaining device does not have a driver installed, it remains unmatched.

Figure 2-2 PCI family objects in the I/O Registry



Writing a Driver for a PCI Bridge

Writing a driver for a PCI bridge is not a simple task. Most PCI bridges do not need any special drivers, and in general, if you are designing the hardware, you are urged to use bridge parts that comply with PCI specifications and thus do not need special drivers. This is not always possible in certain edge cases, however. This is generally because either the hardware already exists or because a transparent bridge does not provide the needed functionality.

This section explains how to write drivers for the most common types of programmable PCI bridges and explains some of the differences between those types of bridges from both a hardware perspective and from a programming perspective.

Types of PCI Bridges

There are three basic types of PCI bridges: PCI host controllers, transparent PCI bus bridges, and nontransparent PCI bus bridges. They each have their own special issues.

- Host controllers—devices that bridge between a processor (host) bus and a PCI bus. These are generally built into a computer and cannot be added after the fact except via a processor-direct slot (PDS). Unless you are adding support for unsupported computers, you should never need to write a driver for a host controller. For more information, see "Writing Drivers for Host Controllers" (page 12).
- Transparent bus bridges—a traditional PCI bridge. These devices are commonly used to extend a PCI bus beyond the physical distance allowed for a simple bus by the PCI specification. Because they connect one standard PCI bus to another standard PCI bus, their interface is well-defined, and they do not need special drivers except to work around bugs in the hardware. For more information, see "Writing Drivers for Transparent Bridges" (page 13).
- Nontransparent bus bridges—a translating PCI bridge, often used to connect two PCI busses that are managed by separate hosts. In some embedded systems, for example, two hosts can use a nontransparent bridge for backplane networking or to share memory between two systems. One side of the bridge is configured (usually in hardware) to be the master. That host sets up the bridge. The two sides may have different address space layouts and different bus address ranges. The bridge then does PCI address translation to rewrite requests as they come through the bridge. For more information, see "Writing Drivers for Nontransparent Bridges" (page 13).
- Other bridges—Other bridges exist, such as CardBus to PCI and PCI to CardBus. Neither of these differs substantially from a transparent PCI bridge except that you need to provide certain additional services such as ejecting and detecting the insertion of cards. Some more extreme bridges, such as ISA to PCI also exist but are rare. You will probably never need to write a driver for any of these sorts of devices, and thus they are not covered here. However, some of the information regarding transparent bridges and host controllers should be a good stating point. For more information, see "Other Bridges" (page 14).

Writing Drivers for Host Controllers

Many of the details of writing a driver for a PCI host controller are device-specific. In many respects, it is similar to writing any other device driver for a device on the main logic board. It is different, however, in that with a PCI host controller you must configure the controller in such a way that the rest of the PCI subsystem can talk to devices behind it. The issues of interrupt assignment and address range assignment are particularly significant in this regard.

The basic process is as follows. The host controller has certain Open Firmware properties (or for Darwin/x86, the platform expert presents certain properties) that identify the physical address and size of its register set. You should map this range into the kernel's virtual address space using an `IOMemoryMap` object.

You then create a PCI address range for the bridge and set the registers on the device appropriately based on the range you obtained. You also need to provide a standard set of routines for communicating with devices on the PCI bus. The system then uses these routines to finish configuring the devices on the bus.

The following methods should be implemented in a PCI host controller driver:

`start`
: Gets information about the bridge (as needed) and maps the bridge hardware into the kernel's address space.

`configure`
: Configures the controller with the base address and size of the PCI bus.

`free`
: Releases any data structures created by the driver, including the mapping of the controller into kernel memory, as well as any locks, queues, and other dynamically allocated structures that the driver may have created during operation or within the `start` routine.

`ioDeviceMemory`
: Returns a pointer of type `IODeviceMemory` to a class instance that contains information about the base address and size of the bus's memory space. This pointer is created in the `start` routine.

`firstBusNum`
: Returns the first PCI bus number that lives beyond this controller.

`lastBusNum`
: Returns the last PCI bus number that lives beyond this controller.

`getBridgeSpace`
: Returns the `IOPCIAddressSpace` entry representing the bridge. This entry contains the bus number, device number, and configuration space bitfield for the bridge itself. The `bits` field, which holds the bitfield, should be set to `0` since it is not relevant except for devices that reside on the PCI bus.

`setConfigSpace`
: Sets a device's configuration bits (bus master enable, memory space enable, and so on) using a write to PCI configuration space, based on information contained in an `IOPCIAddressSpace` entry (bus number, device number, and bitfield).

`configRead32`, `configRead16`, and `configRead8`
: Reads 32, 16, or 8 bytes from an address in PCI configuration space.

`configWrite32`, `configWrite16`, and `configWrite8`
: Writes 32, 16, or 8 bytes to an address in PCI configuration space.

`callPlatformFunction`

Call a method in the platform expert. This method is defined in the `IOPCIBridge` base class, but it may be overridden in an individual PCI bridge driver to allow you to add additional calls specific to your bridge or to override the behavior of existing calls.

For more information, you should study the `AppleMacRiscPCI` driver and the other PCI drivers available from Apple's Open Source website, which can be found at http://www.opensource.apple.com, or contact Apple Developer Technical Support.

Writing Drivers for Transparent Bridges

In general, transparent PCI bridges do not require special drivers, because they obey a very strict specification. However, if a transparent bridge fails to follow the PCI specification, you may need to change the behavior of the transparent bridge driver to support your particular device.

To easily create a driver for a non-compliant transparent bridge, you should subclass the standard Apple transparent bridge driver and modify or extend the basic initialization code. If you do this, be *certain* to set appropriate passive matching parameters or use active matching so that your modified driver matches against only the particular bridge in question. Otherwise, your driver could break the normal operation of other PCI bridges in the system. For details on passive and active matching, see "Open Firmware Matching" (page 16) or read the appropriate parts of *I/O Kit Fundamentals*.

Writing Drivers for Nontransparent Bridges

Fundamentally, nontransparent bridges, which are also often called embedded bridges, are not what most people would describe as bridges, in that they do not generally make devices on one side of the bridge accessible to devices on the other side, nor do they pass traditional interrupts from one side to the other. What makes them bridges is that they are connected to and provide limited communication between two busses.

Drivers for nontransparent bridges are highly device-specific. They appear to each side as a single PCI device, not a bridge. Neither side can see devices on the opposite side of the bridge. They can, however, see address ranges that are explicitly mapped and perform basic interrupt-based communication between the two sides. Thus, it is possible, to a limited degree, to control devices across such a bridge, though they are not generally used in this way.

These devices generally have a series of BATs (block address translation registers) that map a large range onto another large range. In most cases, the client must actually know how to control these BATs whether it is a driver that controls a physical device behind the bridge or a pseudo-device that maps memory across the bridge for backplane networking. You can make the BATs accessible to the client through a user client if the client is in user space, or by simply publishing a nub if the client is another driver within the kernel.

Similarly, these devices often have "doorbell" interrupts for basic communication. These interrupts are triggered by the computer on one side of the bridge and are seen by the computer on the other. Your client interface should reflect this and should provide access as appropriate.

In all other respects, however, nontransparent bridges behave like normal PCI devices from a software perspective. Thus, if you need to write a driver for a nontransparent bridge, you should read "Writing a Driver for a PCI Device" (page 15) for more information.

Other Bridges

Many other types of PCI bridges exist. Most of them fall into one of these categories:

- Bridges to add a non-PCI bus into a system where a PCI bus exists, such as a PCI to ISA bridge
- Bridges to add a PCI bus into a system where a non-PCI bus exists, such as an ISA to PCI bridge
- Bridges between CardBus (a removable PCI standard) and PCI.

With the exception of special features like ejection and device detection, CardBus devices are just PCI devices, and CardBus bridges behave like transparent bridges. Thus, you should read "Writing Drivers for Transparent Bridges" (page 13) and "Writing a Driver for a PCI Device" (page 15) for general information, then look in the `IOPCCardFamily` from Apple's Open Source website for CardBus-specific extensions.

Other bridges, such as those involving ISA or other busses, are beyond the scope of this document. For help with such bridges, you should contact Apple Developer Technical Support.

Debugging a PCI Bridge

Because most Ethernet hardware is found on the PCI bus, debugging a PCI host bridge using traditional Ethernet-based (`gdb`) debugging can pose difficulties. For this reason, you should consider building a custom kernel with `ddb` (serial debugging) support enabled. With this option, you will still be able to debug if the address mappings in your host bridge get inadvertently changed and attaching by Ethernet becomes impossible.

For more information on building a kernel with `ddb` support, and for instructions and advice on debugging drivers using `gdb` and `ddb`, see *Kernel Programming Guide*, available from the Darwin section of Apple's developer documentation website, http://developer.apple.com/Documentation.

Writing a Driver for a PCI Device

A PCI device driver manages a device attached to a PCI bus. The driver matches and attaches to a nub object of the `IOPCIDevice` or `IOAGPDevice` class, configures the device through the PCI configuration space, and sets up the memory-mapped registers or the I/O space to make it possible to control the device. Little else is necessary other than the code to control the device itself.

Matching

A driver that is capable of controlling a device on a PCI bus announces this fact by including a personality in its property list. This personality includes the key `IOProviderClass` with the value `IOPCIDevice`.

Because `IOAGPDevice` is a subclass of `IOPCIDevice`, it is not necessary to match on `IOAGPDevice` for your driver to attach to AGP devices. Thus, you should generally only match on `IOPCIDevice`.

`IOPCIDevice` and `IOAGPDevice` both define two sets of keys that a driver can use for matching, one based on standard PCI registers and another based on Open Firmware. A PCI device driver can use either type of key. It can even use a combination of the two as long as they are in separate personalities.

PCI Matching

PCI device drivers can base their property matching on the PCI configuration space registers for vendor and device ID (offsets 0x00 and 0x02), subsystem vendor and device ID (offsets 0x2C and 0x2E), and class code (offset 0x09). Other registers, such as revision ID and header type, are not available in property matching and must be examined by the `probe` method. The PCI matching dictionary keys are:

Key	Matching behavior
`IOPCIMatch`	Matches against the primary vendor/device ID registers or the subsystem vendor/device ID registers. The primary IDs are checked first; if either of these doesn't match then the subsystem IDs are checked.
`IOPCIPrimaryMatch`	Matches only against the primary vendor/device ID registers.
`IOPCISecondaryMatch`	Matches only against the subsystem vendor/device ID registers.
`IOPCIClassMatch`	Matches against the class code register.

The value for a key can be a single register value or a list of register values separated by spaces. Register values are given as little-endian hexadecimal strings, with the device ID first and the vendor ID second. See "Endianness and Addressing" (page 29) for more information on endianness.

In addition to the value itself, each key value can include a mask indicating that only part of the value should be compared.

Here are some examples of matching dictionary entries. The first example matches a PCI device with a primary vendor ID of 0x8086 and primary device ID of 0x1229, or with subsystem IDs of those values.

```
<key>IOPCIMatch</key>
    <string>0x12298086</string>
```

The second example matches a PCI device with a primary vendor ID of 0x8086 and a primary device ID of 0x1229 or 0x1227, or with subsystem IDs of those values.

```
<key>IOPCIMatch</key>
    <string>0x12298086 0x12278086</string>
```

The third example matches a PCI device with a primary vendor ID of 0x8086 and a primary device ID of 0x1229. Subsystem IDs aren't checked.

```
<key>IOPCIMatch</key>
    <string>0x12298086</string>
```

The fourth example uses a mask to match a PCI device with a primary vendor ID of 0x8086 and any primary device ID whose first two digits are 0x12. Subsystem IDs aren't checked.

```
<key>IOPCIPrimaryMatch</key>
    <string>0x12008086&amp;0xFF00FFFF</string>
```

The mask is attached to the register value using an ampersand character, which must be encoded as `&` in XML if you are editing it directly. If you are using Xcode or Property List Editor, the `&` escape is represented simply as an ampersand (&).

Note: In the register value, bits that are set to be ignored by the mask must currently be set to 0.

This entry matches a PCI device with a primary vendor ID of 0x8086 and a class code beginning with 0x0200, the class code for Ethernet controllers.

```
<key>IOPCIMatch</key>
    <string>0x00008086&0x0000FFFF</string>
<key>IOPCIClassMatch</key>
    <string>0x02000000&0xFFFFFF00</string>
```

Note: The `IOPCIClassMatch` value currently requires a full 4-byte value for matching, of which the last byte is ignored.

Open Firmware Matching

A PCI device driver can use Open Firmware matching in a personality by specifying a value for the `IONameMatch` key. The value is either a single string or an array of strings, which are compared against the values for the Open Firmware device properties `name`, `compatible`, `device_type`, or `model`.

If any of the values for the `IONameMatch` key match any of the Open Firmware properties, the match is considered a success and an instance of the driver is created for the personality. The name that resulted in a match is stored as a property in the driver's I/O Registry entry in the `IONameMatched` key (`kIONameMatchedKey`).

For more information on name matching, you should consult the documentation for the `IOService` class.

PCI Device Inspection and Configuration

Before a driver begins interacting with a PCI device, whether in the `probe` method or the `start` method, it typically examines the values in the PCI configuration space registers and sets some of them as required. The IOPCIDevice class defines several methods and register masks to make doing so easy.

The most general methods are `configRead32`, `configWrite32`, and `setConfigBits`. The first method reads a full 32-bit value from a 4-byte aligned address in the PCI configuration space. The driver can then extract the specific register value. A driver might use this method during probing to examine the value of the revision ID register, for example. `IOPCIDevice` defines a number of constants for the various register offsets.

The method `configWrite32` writes a full 32-bit value into a 4-byte aligned address in the PCI configuration space. The more flexible method `setConfigBits` can write individual bits into the configuration space without disturbing others. It takes a 32-bit value to write along with a 32-bit mask indicating which bits to set or clear. This is commonly used to write to the command register. This code fragment, for example, enables the memory write and invalidate (MWI) transaction on the bus:

```
UInt32 configMask = 0;
UInt32 configBits = 0;

configMask = configBits = kIOPCICommandMemWrInvalidate;

// provider is the PCI nub object
provider->setConfigBits(kIOPCIConfigCommand,
    configMask, configBits);
```

For commonly used configuration settings, IOPCIDevice defines convenience methods that take a `bool` argument for enabling and disabling the settings. These methods are `setMemoryEnable` for memory-mapped control of the PCI device and `setIOEnable` for I/O-mapped control.

Some devices require bus mastering to work properly. For most cards, Open Firmware should set this up automatically based on information in the card's declaration ROM before Mac OS X even starts to boot. However, this automatic setup does not always occur, depending on the specifics of the particular card and on decisions made by the card vendor when populating the card with ROMs. If a card is not configured correctly in this regard, you can use `setBusMasterEnable` to manually turn bus mastering on or off for the device.

Other methods for inspecting the PCI device include `findPCICapability`, `getBusNumber`, `getDeviceNumber`, and `getFunctionNumber`. For details on these and all other methods, see the reference documentation in `/System/Developer/Documentation`.

Device Access

The `start` method is where a driver sets up its hardware for operation. For PCI devices this involves configuring the hardware as described above, setting up the communication channel between the CPU and the device, and then performing whatever device-specific initialization is required. You can gain access to PCI device by mapping the device's registers (its memory space) into the kernel's address space, or by reading and writing in the device's I/O space using the nub object.

Mapping the device registers is the most convenient way to set up device control. Once this is done, the driver has very little further interaction with the `IOPCIDevice` nub. Instead, it interacts with the hardware as if it were ordinary memory, albeit uncached memory, using either device accessor methods for byte swapping or by directly dereferencing pointers into the card's memory space.

I/O space access provides a single programming interface for using I/O space. On processors with a separate I/O memory bus, the I/O space methods actually use this bus. On processors without a separate I/O memory bus, the methods simply use memory-mapped space and handle the required synchronization automatically.

Memory-Mapped Device Access

Setting up memory-mapped device control is a two-step process. The driver must first get an `IOMemoryMap` object for one of the device's base address registers, using the `mapDeviceMemoryWithRegister` method. Constants are available for the base address register offsets. The driver then calls the memory map's `getVirtualAddress` method to retrieve a pointer to the memory-mapped register block in the kernel's virtual address space. Here's an example of the standard idiom for setting up memory-mapped registers, which is typically done in the driver's `start` method:

```
IOMemoryMap *map;

map = provider->mapDeviceMemoryWithRegister(kIOPCIConfigBaseAddress0);
if (!map) {
    // Handle error
}
deviceRegisterPointer = (deviceStruct *)map->getVirtualAddress();
```

Both `map` and `deviceRegisterPointer` are instance variables of the driver. Because the driver created the map object by invoking `mapDeviceMemoryWithRegister`, it must cache the pointer and later release the map object in its `stop` method. `deviceRegisterPointer` is a pointer to a structure defined by the driver that corresponds to the layout of the device-specific registers.

You should note two things when using memory-mapped control of a PCI device. First, the PCI bus standard uses little-endian byte addressing. The endianness of the bus generally has minimal impact on device driver writers. However, the endianness also encourages a related model of register layout that directly affects most driver writers. For more information, see "Endianness and Addressing" (page 29).

Second, while accesses into PCI space are explicitly uncached, in-order execution of these accesses is not guaranteed. When order of execution is important, I/O barriers may be used.

I/O barriers are often used for the following reasons:

- To ensure that the operands of a command sent to a device are written into the device's registers before stating the command.

- To ensure that a command has started executing before polling for its completion status.
- To ensure that an interrupt line is cleared before waiting for that interrupt again.

A general-purpose I/O barrier in Mac OS X is provided by the `libkern` function `OSSynchronizeIO`. On the PowerPC architecture, for example, this issues an `eieio` instruction to enforce in-order execution of the write operation. Because of the performance impact, you should only use an I/O barrier when in-order execution is required for correct operation.

Synchronization is required only when writing directly to hardware registers; when setting up buffers and structures used by hardware, it is not necessary. See an example PCI driver from Apple's open source collection for examples of how to use `OSSynchronizeIO`.

In unusual circumstances, you might find it desirable to enable caching on portions of I/O space. If you do this, then writes to that portion of PCI space are no longer guaranteed to be written back to the device's memory space. Additional synchronization is necessary in these cases. Enabling caching and performing I/O to cached I/O spaces is beyond the scope of this book. If you need such support you should contact Apple Developer Technical Support for additional information.

I/O Space Device Access

Setting up I/O space access is required only for devices with relocatable I/O spaces. In this case, the driver must obtain an IOMemoryMap object, just as for memory-mapped access. Apart from this, however, I/O space access is performed using a set of six methods for reading and writing 32-bit, 16-bit, and 8-bit values. The `ioWrite16` method, for example, takes a byte offset into the memory space, the value to write, and the memory map for the I/O space. For devices with nonrelocatable I/O spaces, the memory map argument is omitted. The I/O space methods perform all necessary byte swapping and synchronization. For more information on these methods, see the IOPCIDevice reference documentation in `/System/Developer/Documentation/Kernel/IOKit/IOPCIDevice`.

Writing a Driver for an AGP Device

AGP is a superset of PCI providing additional functionality that is optimized for video devices. This chapter describes this added functionality at a high level. For detailed programming information, you should consult the documentation for the `IOAGPDevice` family, which can be found on the Apple Developer Documentation website (http://developer.apple.com/Documentation).

Fundamentally, supporting an AGP device does not require much more effort than supporting a PCI device. PCI drivers should "just work" for AGP devices as-is with the possible addition of a new matching personality if the device has a different ID or Open Firmware name. You can add an additional personality for AGP in exactly the same way as you would add a personality for a standard PCI device, as described in "Matching" (page 15).

You should note, however, that a standard PCI driver will not see AGP's performance gains without additional driver changes to facilitate the use of AGP memory transactions to and from the device.

Important: Before attempting to write a driver for an AGP device (and particularly a driver that uses AGP's special features), you should contact Apple Developer Technical Support.

To add this support, you must first detect whether the device your driver matched against is, in fact, an AGP device. You can do this by trying to cast your provider to the type `IOAGPDevice` by calling `OSDynamicCast`. If this call succeeds, your driver matched against an AGP device. If it fails, your driver matched against a non-AGP PCI device such as a standard PCI card or a CardBus card.

Once you have determined that your driver has matched against an AGP device, you should use the `IOAGPDevice` method `createAGPSpace` to create an AGP space for communication with the device and enable AGP transactions on the bus. You should use the method `destroyAGPSpace` when your driver unloads.

From there, you can use the method `commitAGPMemory` to make blocks of memory accessible to your device via AGP calls—for example, if your graphics card needs to read a texture out of main memory. You should use `releaseAGPMemory` to free the AGP mappings for these memory regions when you're finished with them.

With those changes, the PCI device driver should reflect the added performance benefits that AGP provides without further modification.

For more information on AGP devices, consult the documentation for the `IOAGPDevice` family and the graphics acceleration SDK.

Taking Primary Interrupts

Most device drivers never need to take primary interrupts because their interrupt lines don't cascade into the system's interrupt controller. For example, FireWire and USB devices have a notion of interrupts, but are really just messages on a serial bus. These are commonly referred to as software interrupts because apart from the interrupt caused by the message itself, the interrupt is entirely simulated in software.

PCI devices, however, are supported by hardware interrupts. A physical line runs between the PCI slot and the PCI controller, which routes that line to an interrupt input on the interrupt controller, which presents the value of that line as a bit value in a register and raises the processor's interrupt line.

You can take a hardware interrupt in two ways: directly via an interrupt handler (primary interrupt) and indirectly via an interrupt service thread (secondary interrupt). The second method is strongly preferred, as it avoids generating excessive interrupt latency for other devices on the system. However, the first method can be used to guarantee low latency operation when necessary.

Interrupts and the Device Tree

Open Firmware automatically assigns interrupts to PCI devices. Typically a PCI device has only a single interrupt, but it is possible to have PCI devices that generate more than one interrupt (for example, a device interrupt and a DMA interrupt).

In any case, all of the interrupts used by the device are listed in an array as part of the device's I/O Registry entry. Like a C array, the first interrupt is at offset 0, the second at offset 1, and so on.

When registering to receive an interrupt, you must specify the offset into the interrupt list for the interrupt you want to receive. If you need to receive notification for multiple interrupts, you must enable them each individually.

In the unlikely event that Open Firmware should fail to allocate interrupts correctly for your device, you should contact Apple Developer Technical Support for assistance, as correcting problems with interrupt allocation is beyond the scope of this book.

What is Interrupt Latency?

Interrupt latency refers to the amount of time between when an interrupt is triggered and when the interrupt is seen by software. This can be caused by many factors. Some of these include:

- other drivers keeping interrupts disabled for a long period
- blocking in the interrupt service thread
- scheduling latency (waiting for the interrupt service thread to be scheduled)
- interrupt service thread priority (particularly for non-kernel drivers)

If you are writing device drivers, it is your responsibility to avoid causing too much latency for other devices in the system. In extreme cases, it is possible to actually cause interrupts to be dropped by leaving interrupts turned off for too long. This can cause system instability, and in some machines, can actually cause the computer to spontaneously power down by causing a PMU synchronization failure.

The best way to avoid such problems is to use interrupt service threads. This is the preferred way to service interrupts. However, if your driver is well-behaved and doesn't spend huge amounts of time copying data, it is also possible to write reasonable device drivers that operate directly in an interrupt context. This is briefly discussed in "Taking Interrupts Directly" (page 26).

Interrupts via Interrupt Service Threads

Interrupt service threads are the standard way of handling interrupts in Mac OS X device drivers, though they often go by another name, the work loop.

A work loop is a thread whose sole purpose is to wait for an event such as an interrupt to occur, then call an appropriate handler function to do the actual work of processing the interrupt—checking the result, copying data, and so on. This activity occurs in a kernel thread.

The kernel thread does not receive the actual interrupt, however. A low-level interrupt handler receives the primary interrupt. It then generates a software interrupt known as a secondary interrupt. The interrupt service thread receives that secondary software interrupt. Thus, routines running in an interrupt service thread do not have to obey the same rules as an actual interrupt handler; they can block, call `IOLog`, and so on.

The following code is an example of registering to receive two (secondary) interrupts using a work loop:

```
/* class variables */
IOWorkLoop *myWorkLoop;
IOFilterInterruptEventSource *interruptSource;
IOFilterInterruptEventSource *DMAInterruptSource;

/* code in start() */
myWorkLoop = IOWorkLoop::workLoop();

if( myWorkLoop == NULL) {
    IOLog( "org_mklinux_iokit_swim3_driver::start: Couldn't "
        allocate workloop event source\n");
    return false;
}

interruptSource = IOFilterInterruptEventSource::filterInterruptEventSource(
    (OSObject*)this,
    (IOInterruptEventAction)&org_mklinux_iokit_swim3_driver::handleInterrupt,
    (Filter)&org_mklinux_iokit_swim3_driver::filterInterrupt,
    (IOService*)provider,
    (int)0 );

if ( interruptSource == NULL ) {
    IOLog( "org_mklinux_iokit_swim3_driver::start: Couldn't"
        "allocate Interrupt event source\n" );
    return false;
}

if ( myWorkLoop->addEventSource( interruptSource ) != kIOReturnSuccess ) {
```

```
    IOLog( "org_mklinux_iokit_swim3_driver::start - Couldn't add Interrupt"
        "event source\n" );    return false;}

DMAInterruptSource = IOFilterInterruptEventSource::
    filterInterruptEventSource(
        (OSObject*)this,
        (IOInterruptEventAction)&org_mklinux_iokit_swim3_driver::
            handleDMAInterrupt,
        (Filter)&org_mklinux_iokit_swim3_driver::filterDMAInterrupt
        (IOService*)provider,        (int)1 );

if ( DMAInterruptSource == NULL ) {
    IOLog( "org_mklinux_iokit_swim3_driver::start: Couldn't"
        "allocate Interrupt event source\n" );
    return false;
    }

if ( myWorkLoop->addEventSource( DMAInterruptSource ) != kIOReturnSuccess )
    {
    IOLog( "org_mklinux_iokit_swim3_driver::start - Couldn't add "
        "Interrupt event source\n" );
    return false;
}

myWorkLoop->enableAllInterrupts();
```

The methods `handleInterrupt` and `handleDMAInterrupt` will now be called for the first and second device interrupts (offsets 0 and 1), respectively.

Notice the methods `filterInterrupt` and `filterDMAInterrupt`. These methods are called upon receipt of a hardware interrupt, and should do the minimum amount of work necessary to determine whether an interrupt belongs to your driver or not.

This use of interrupt filters is strongly recommended over `IOInterruptEventSource` to ensure that your driver performs as well as possible should your device end up sharing an interrupt with another device. If you do not write a filter function, the system will have to call each driver that registers for a given shared interrupt one-by-one, resulting in unpleasant latency for all drivers involved.

If the interrupt belongs to your device, the filter method should return `true`. Otherwise it should return `false`. Because the two interrupts (device and DMA) above could, in theory, be shared on the same interrupt line, additional care should be taken to determine the nature of the interrupt, and to return `true` only for the right kind of interrupt—that is, `filterInterrupt` should return `true`*only* for a device interrupt, and `filterDMAInterrupt` should return `true`*only* for a DMA interrupt.

Warning: Interrupt filter functions run in a primary (hardware) interrupt context. This means that they have additional restrictions on what they are allowed to do, and that they *must* be extremely efficient.

For more information on the restrictions that apply to filter functions, see "Taking Interrupts Directly" (page 26).

If your interrupt filter returns `true`, your interrupt handler routine will be started on your work loop automatically by the I/O Kit. The interrupt will remain disabled in hardware until your interrupt service routine completes.

In some cases, such as pseudo-DMA, this behavior may not be what you want. For this reason, you may elect to have your filter routine schedule the work on the work loop itself, then return `false`. If you do this, the interrupt will not be disabled in hardware, and thus, you could receive additional primary interrupts before your work-loop–level service routine completes. Because this method has implications on synchronization between your filter routine and interrupt service routine, you should generally avoid doing this unless your driver requires pseudo-DMA.

For additional information, consult the document *I/O Kit Fundamentals* and the `IOFilterInterruptEventSource` class documentation in the device drivers API reference, available from the Apple Developer Documentation website.

Taking Interrupts Directly

This method is generally not advisable, as it can cause undesirable behavior including high interrupt latency for other drivers, audio stuttering, lost interrupts, and even erratic overall system behavior.

However, if you need exceptionally low latency for certain specialty devices, it may be necessary to take primary interrupts directly. You should remember that drivers that take primary interrupts must not block in their handler routines, which means that many calls are not allowed, including `IOLog`.

You can register a handler for a raw interrupt in one of two ways: using an `IOFilterInterruptEventSource` or by using `registerInterrupt`.

If you are sharing an interrupt between multiple devices, you should use an `IOFilterInterruptEventSource` handler.

If you are writing a driver whose interrupt handler must run partially in a raw interrupt context, you can use `registerInterrupt`. You should *only* consider doing processing that is *extremely* fast in a primary interrupt context. In general, if the amount of work done takes less time than a context switch, it is acceptable to do it in a primary interrupt context.

Regardless, you should do *only* the low-latency portions in an interrupt context and defer any heavy lifting to an interrupt service thread.

Taking a Primary Interrupt Using IOFilterInterruptEventSource

For information on IOFilterInterruptEventSource, see "Interrupts via Interrupt Service Threads" (page 24).

Taking a Primary Interrupt Using registerInterrupt

Taking a primary interrupt in this fashion is dangerous. You should not ever do this unless you have determined that it is impossible to get adequate latency using a secondary interrupt handler. Before proceeding, you should contact Apple Developer Technical Support for guidance.

When working in the primary interrupt context, nearly all calls are unsafe. Even things like `IOLog` can block, and blocking in a primary interrupt context will result in a kernel panic (since it would otherwise result in an unexplained hang). Only Mach simple locks (spinlocks) are safe in this context, since they disappear in a uniprocessor environment.

You should be extremely careful to avoid keeping interrupts turned off for an extended period of time. For example, multi-millisecond polling, delays, and copying large amounts of data are not acceptable in a primary interrupt context. Such activity should always be deferred to an interrupt service thread.

With those caveats in mind, the sample code below shows how to register a handler for a primary (hardware) interrupt directly without using work loops. The first example shows how to register a C++ class member function. The second example shows how to register an ordinary C function.

```
provider->registerInterrupt(0, this,
        OSMemberFunctionCast(IOInterruptAction, this,
            &MyClass::handleInterrupt), 0);
provider->enableInterrupt(0);
provider->registerInterrupt(1, this,
        (IOInterruptAction) &handleDMAInterrupt, 0);
provider->enableInterrupt(1);
```

As with the previous example, when the first interrupt (offset 0) occurs, the function `handleInterrupt` will be called, and when the second interrupt (offset 1) occurs, the function `handleDMAInterrupt` will be called.

Endianness and Addressing

The subject of endianness, or byte order, is rarely fully explained in the context of mixed-endian environments. This chapter explains the various issues involved in endianness as it relates to PCI on PowerPC-based and Intel-based Macintosh hardware.

What is Endianness?

In Jonathan Swift's *Gulliver's Travels*, the Lilliputians were split into two factions over the matter of which end of an egg to break open when consuming it. Big Endians broke their egg on the large end, while Little Endians broke it on the small end. In much the same way, computer designers are in a constant state of disagreement over which end of a multi-byte value to consume first.

For simplicity, endianness in 32-bit numbers is often represented by the letters ABCD, where the letters are used to describe the order in which the most, second to most, second to least, and least-significant bytes are stored in memory.

Little endian refers to storing the little end of a multibyte value at the lowest address. The number 513 in a 32-bit value, for example, is stored as `0x01020000` (DCBA), with the `02` representing 512, and the `01` representing the other 1. The address pointer for the start of the value points at the `01`.

Big endian refers to storing the big end of a multibyte value first. This is the order in which you are probably accustomed to seeing numbers represented in print. The number 513 is represented as `0x00000201` (ABCD), and an address pointer for the start of this value points to the leftmost `00`.

To be pedantic, there are actually *n!* byte orders where n is the number of bytes in a word on a given architecture, and thus there can be arbitrarily many byte orders if you consider infinitely long word sizes.

A few arcane systems once represented 513 as `0x00000102` (BADC). This is equivalent to storing a 32-bit word as two little endian 16-bit words. In theory, CDAB ordering would also be possible, as would twenty other possible byte orders. Fortunately, though, it is unlikely that you will ever encounter a system whose byte ordering is anything other than ABCD (big endian) or DCBA (little endian).

On the Macintosh platform, PowerPC-based Macintosh computers use big endian addressing, while Intel-based Macs use little-endian addressing.

Byte-Invariant Addressing vs. Data Structure Order

PCI busses are, by design, little endian. Most non-Intel-based computers are big endian. This presents a number of issues for hardware designers and programmers alike. These issues mostly deal with munging data between the PCI bus and the rest of the system.

There are two fundamental ways in which data must be transformed when dealing with PCI devices: changing the byte order of the data itself and preserving byte-invariant ordering. These are two separate issues and must not be confused.

Byte-invariant addressing is a property of the bus bridge itself. What byte-invariant addressing means is that when you access a PCI device's address space byte by byte, you obtain the data in the order in which it is stored in the device's memory.

From the software designer's perspective, this means that the hardware does not byte swap the data. However, from the hardware designer's perspective, the hardware must byte swap *all* data. This difference in perspective is explained in "Natural Byte Order and Preserving Byte Invariant Addressing" (page 30).

Data structure order, however, refers to the order in which the bytes of a multibyte number are stored in the card's memory. If the data structure order is different than the byte order of the host machine, additional byte-swapping must be done in software. This is described in "Data Structure Order" (page 30).

Natural Byte Order and Preserving Byte Invariant Addressing

From a hardware point of view, all data on a PCI bus is little endian. However, on PowerPC hardware, the main CPU bus is big endian. The physical ordering of bytes on the two busses are actually opposite. If data were read from a PCI bus using a big endian processor *without* any manipulation, the bytes would be reversed in 32-bit chunks (or 64-bit chunks for 64-bit PCI transactions).

The effect of such a reordering would be that address zero would correspond to address three on the opposite side of the bridge, address one would correspond to address two, and so on. This is clearly not desirable.

Instead, at the hardware level, most PCI host bridge chips can swap the byte order. This ensures that per-byte addresses on one side of the bridge are equal to the per-byte addresses on the other side of the bridge.

Thus, if you are writing a driver for a host bridge chip on PowerPC-based computers or other big-endian architectures, you probably want to turn automatic byte swapping on to ensure byte-invariant addressing. However, if you are writing a driver for a host bridge chip on Intel or other little-endian architectures, you probably do not want to do this swapping.

You should note, however, that from a device driver point of view, byte 0 is still byte 0 and byte 1 is still byte 1. Thus, you shouldn't refer to this process as byte swapping. This swapping process is more accurately called preserving byte-invariant addressing or preserving natural byte order.

Data Structure Order

Data structure order is what most developers think of when they hear the phrase "byte swap." PCI devices may lay out their registers in any way that the card designer sees fit. In most cases, the register map for a card contains some combination of 1-byte, 2-byte, and often 4-byte registers. Each of these registers can be big or little endian. For most devices, registers are little endian, but this is not guaranteed.

When accessing a device register, you generally must know the register's address (usually as an offset from the start of the card's memory space), the register's endianness, and the register's size. For single byte addresses, you need to know only the address and size, since endianness only affects multibyte values.

The basic rules are straightforward:

- If a register's order is specified as little endian—that is, if the byte with the lowest offset is defined in the device specifications as being the least significant byte (LSB)—then you should store data into it using a byte-reversing function on PowerPC-based computers, or a non–byte-reversing function on Intel architecture systems.
- If a register's order is specified as big endian—that is, if the byte with the lowest offset is defined in the device specifications as being the most significant byte (MSB)—then you should use a byte-reversing function on Intel systems and a non–byte-reversing function on PowerPC systems.
- All writes to PCI configuration space are little endian. However, for configuration space changes, you should use special configuration space functions, rather than generic byte-swapping functions.

In short, the hardware byte swapping ensures that the byte order as seen by software is the same as the byte order specified in the chip documentation for the device. If the device registers are big endian, store values as big endian. If the device registers are little endian, store values as little endian.

Of course, manual byte swapping is tedious at best, error-prone at worst, and generally unpleasant on the whole. For this reason, several functions and methods exist in Mac OS X to facilitate reading and writing registers of various sizes in both little and big endian modes.

For operations on PCI configuration space, the PCI framework provides methods such as `configRead16` and `configWrite16`.

The `libkern` byte-access functions, such as `OSReadLittleInt16` and `OSWriteLittleInt16`, provide generic byte swapping for data based on the size and endianness of the register being read or written. Similar functions, such as `OSReadBigInt16` and `OSWriteBigInt16`, are available for big-endian device registers, in the rare event that you might run into such a register.

These functions, found in `libkern/OSByteOrder.h` perform byte swapping if the host memory ordering is not the same as the register's byte order. You should always write drivers using these platform-independent macros and functions rather than using explicit swap functions like `OSWriteSwap16` directly.

For more information, see the header file for these functions, `/System/Library/Frameworks/Kernel.framework/Headers/libkern/OSByteOrder.h`, and any of the example PCI drivers.

Working With Addresses

Many people mistakenly assume that addresses cannot be treated like other data because of byte ordering issues. In fact, you can treat addresses in the same way as you treat any other data.

When accessing addresses on the PCI bus from software, addresses are translated as needed within the bridge, just as any other data; thus the bytes of an address are stored in the same byte-invariant fashion in a device's registers as they appear on the CPU side of the bridge. However, when addresses are handed off to a PCI device that operates on those addresses, you must take care.

Generally, registers on PCI devices are in little-endian order, while addresses (at least on PowerPC and other big-endian architectures) are stored in big-endian order. Thus, they must be byte swapped. Just like any other data, though, if a PCI device uses big-endian ordering for that register, you should not byte swap. The reverse applies on Intel and other little-endian architectures.

In other words, treat addresses just as you would treat any other data. Addresses should be stored as little endian when storing them into little-endian registers, and big endian for big-endian registers.

Document Revision History

This table describes the changes to *Writing PCI Drivers*.

Date	Notes
2006-04-04	Fixed typographical error in code sample.
2006-01-10	Corrected typographical errors.
2005-10-04	Fixed some example errors, reformatted code examples, and clarified some wording now that Intel-based hardware support is no longer Darwin-specific.
2004-11-02	Minor typographical fixes.
2004-08-31	Added information on `IOFilterInterruptEventSource` as it relates to interrupt blocking.
2003-04-01	Filled in missing material about PCI bridges and AGP devices. Added references to Designing PCI Cards and Drivers. Added a chapter about endianness and how it differs from byte-invariant addressing. Added revision history.

Document Revision History