Designing PCI Cards and Drivers for Power Macintosh Computers

Revised Edition

Revised 3/26/99

Technical Publications

Contents

Figures, Tables, and Listings

About This Book

This revision to *Designing PCI Cards and Drivers for Power Macintosh Computers* includes corrections to the material in the original document and incorporates information that PCI card and device driver developers need to know about the NewWorld architecture introduced in the iMac computer and incorporated in Macintosh computers released after the iMac.

The book describes the Macintosh implementation of the Peripheral Component Interconnect (PCI) local bus established by the PCI Special Interest Group. It also describes the Macintosh Open Firmware model, and provides the definitive reference for Power Macintosh native device drivers and the Macintosh Name Registry.

The PCI local bus standard defines a high-performance interconnection method between plug-in expansion cards, integrated I/O controller chips, and a computer's main processing and memory system.

The first generation of Power Macintosh computers—the Power Macintosh 6100, 7100, and 8100 models—supported NuBus™ expansion cards. Subsequent Power Macintosh models support the PCI standard. This book contains useful information for developers who want to design PCI expansion cards and their associated software to be compatible with Macintosh computers.

The information about PowerPC native device drivers, the Macintosh Name Registry, and the Driver Services Library in Part 4 of this document is not specific to PCI cards. The reference material found in those chapters is useful to anyone interested in writing device drivers for peripherals connected to Power Macintosh computers released after the PCI bus architecture became part of the Macintosh hardware design.

This book is general and does not provide model specific details. You should refer to the developer notes that accompany each Macintosh product release for exact details of that product's PCI implementation.

This document is written for professional hardware and software engineers. You should be generally familiar with existing Macintosh technology, including Mac OS (the Macintosh system software) and the Apple RISC technology based on the PowerPC microprocessor. For recommended reading material about Macintosh and PowerPC technology, see the documents listed in "Supplementary Documents" (page 26).

Contents of This Book

This book is divided into three parts and contains 15 chapters.

PCI Bus Overview

Part 1, "The PCI Bus," describes the PCI bus and tells you how it works with Power Macintosh computers:

- Chapter 1, "PCI Bus Overview," describes the PCI standard and summarizes the ways that Power Macintosh computers comply with it.
- Chapter 2, "Data Formats and Memory Usage," defines the formats in which data moves over the PCI bus and the memory spaces reserved for PCI use.

System Startup by Open Firmware

Part 2, "The Open Firmware Process," describes the startup process in Power Macintosh computers that support devices attached to the PCI bus and run Mac OS. The first chapter in Part 2 describes how the Macintosh boot process has changed and how ROM and memory mapping differ since the introduction of the iMac computer. This is essential reading for PCI card and peripheral device developers interested in participating in the boot process on the latest Macintosh computer models.

- Chapter 3, "Introduction to the NewWorld Architecture," describes how the NewWorld Architecture works from an organizational and execution flow standpoint and describes differences from older architectures.
- Chapter 4, "Startup and System Configuration," describes how PCI-compatible Macintosh computers recognize and configure peripheral devices connected to the PCI bus.
- Chapter 5, "PCI Open Firmware Drivers," discusses Open Firmware drivers, which control PCI devices during the Open Firmware startup process.

Native PowerPC Drivers

Part 4, "Native Device Drivers," tells you how to design and write runtime PCI card drivers for Power Macintosh computers. These drivers are called *native* because they are written for execution by the native instruction set of the PowerPC microprocessor. Part 3 consists of these chapters:

- Chapter 6, "Native Driver Overview," presents the general concepts and framework applicable to PCI drivers for PowerPC Macintosh computers.
- Chapter 7, "Finding, Initializing, and Replacing Drivers," discusses what PCI driver and card designers can do to improve the compatibility of their products.
- Chapter 8, "Writing Native Drivers," gives you details of native driver design and coding, including how to use services provided by the Macintosh Driver Loader Library.
- Chapter 9, "Driver Loader Library," describes the Driver Loader Library (DLL), a CFM shared-library extension to the Macintosh Device Manager.
- Chapter 10, "Name Registry," describes the Mac OS data structure that stores device information extracted from the PCI device tree.
- Chapter 11, "Driver Services Library," details the general support that Mac OS provides for device drivers, including interrupt and timing services.
- Chapter 12, "Expansion Bus Manager," discusses a collection of PCI bus-specific system services available to native device drivers.
- Chapter 13, "Graphics Drivers," describes the calls serviced by typical display drivers.
- Chapter 14, "Network Drivers," describes the construction of a sample network driver.
- Chapter 15, "SCSI Drivers," describes the construction of a sample native SCSI Interface Module (SIM) compatible with Macintosh SCSI Manager 4.3.

Appendixes

Four appendixes follow the main part of this book.

- Appendix A, "Big-Endian and Little-Endian Addressing," discusses the theory and problems of handling mixed-endian formats.

- Appendix B, “Graphic Memory Formats,” describes the ways that graphic information and video frames are stored in PCI-based Power Macintosh computers.
- Appendix C, “PCI Header Files,” describes the PCI header files and lists all the routines and data structures documented in this book.
- Appendix D, “Abbreviations,” lists the abbreviations and acronyms used in this book.

Supplementary Documents

The documents described in this section provide information that complements or extends the information in this book.

Apple Publications

Apple publishes a variety of books, software development kits (SDKs), and technical notes designed to help third-party developers design hardware and software products compatible with Apple computers. Apple publications and SDKs can be found on the Developer CD Series discs and on the Apple Developer website at

http://www.apple.com/developer/

For the latest documentation related to PCI and Open Firmware development, you should visit

http://bootrom.apple.com/

Other Publications

This book cites several documents that are not published by Apple. They are available from the organizations listed below.

American National Standards Institute

ANSI has prepared a standard called *ANSI/IEEE X3.215-199x Programming Languages—Forth.* It is a useful reference for the Forth language used in the Open Firmware process. You can contact ANSI at

American National Standards Institute
11 West 42nd Street
New York, NY 10036
Phone 212-642-4900
Fax 212-302-1286

http://web.ansi.org/

FirmWorks

FirmWorks has issued a book, *Writing FCode Programs for PCI,* that provides essential information for programmers designing Open Firmware drivers for PCI cards. This book is published by FirmWorks and is available by writing to

FirmWorks
480 San Antonio Road, Suite 230
Mountain View, CA 94040-1218
Email info@firmworks.com
Phone 415-917-0100
Fax 415-917-6990

http://www.firmworks.com/

Institute of Electrical and Electronic Engineers

The essential IEEE document for designers of Macintosh-compatible PCI card firmware is *1275-1994 Standard for Boot (Initialization, Configuration) Firmware, IEEE part number DS02683.* It is referred to in this book as *IEEE Standard 1275*. You can order it from

IEEE Standards Department
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331
Phone 800-678-4333 (U.S.)
Phone 908-562-5432 (International)

http://standards.ieee.org/index.html

Note
The P1275 Working Group continues to work on new PCI bus and processor bindings, as well as extensions to *IEEE Standard 1275.* Current documents, including *PCI Bus Binding to IEEE 1275-1994,* are available on an anonymous Internet FTP site, donated by Sun Microsystems, at http://playground.sun.com/pub/p1275. ◆

PCI Special Interest Group

The essential PCI standard document for designers of Macintosh-compatible PCI cards is *PCI Local Bus Specification,* Revision 2.0. It is available from

PCI Special Interest Group
P. O. Box 14070
Portland, OR 97214
Phone 800-433-5177 (U.S.)
Phone 503-797-4207 (International)
Fax 503-234-6762

http://www.pcisig.com/

The PCI SIG also publishes *PCI Multimedia Design Guide* and the *PCI to PCI Bridge Architecture Specification.*

SunSoft Press

SunSoft Press has issued a book, *Writing FCode Programs,* that provides useful background information about FCode. Its ISBN number is 0-13-107236-6. This book is published by PTR Prentice Hall and is available at most computer bookstores.

Conventions and Abbreviations

This book uses the following typographical conventions and abbreviations.

Typographical Conventions

New terms appear in **boldface** where they are first defined. These terms also appear in the glossary (page 601).

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

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

Notes

The following three types of notes in this book are set apart from the text:

Note
A general note like this contains information that is interesting but not essential for an understanding of the subject. ◆

IMPORTANT
Important notes call your attention to information that you should not ignore. ▲

▲ **WARNING**
Warnings tell you about potential problems that could result in system failure or loss of data. ▲

Abbreviations

Wherever possible, this book uses standard abbreviations for units of measure. It also supports readability by using acronyms for many technical terms. Appendix D, “Abbreviations,” contains a complete list of the abbreviations and acronyms used in this book.

P A R T   O N E

The PCI Bus

This part of *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the PCI bus and tells you how it works with Power Macintosh computers. It contains three chapters:

- Chapter 1, “PCI Bus Overview,” describes the PCI standard and summarizes the ways that Power Macintosh computers comply with it.
- Chapter 2, “Data Formats and Memory Usage,” defines the formats in which data moves over the PCI bus and the memory spaces reserved for PCI use.

Later parts of this book cover the following topics:

- Part 3, “The Open Firmware Process,” describes the startup process in Power Macintosh computers that support the PCI bus. This includes information about the startup process for the NewWorld architecture as well as earlier architectures. Part 2 begins on (page 67).
- Part 4, “Native Device Drivers,” tells you how to design and write run-time native device drivers that support the PCI-bus compatible Power Macintosh architecture. Part 4 begins on (page 137).

C H A P T E R 1

PCI Bus Overview

The **PCI local bus** standard defines a method for connecting both ASIC chips and plug-in expansion cards to a computer's main memory and processing circuitry. Power Macintosh computers use the PCI bus or buses to communicate both with internal I/O chips and with plug-in expansion cards. This book discusses Apple's implementation of the PCI bus for expansion cards.

Apple's underlying policy is to support the PCI standard, as expressed in *PCI Local Bus Specification,* Revision 2.1, referred to here as the **PCI specification.** This standard specifies the logical, electrical, and mechanical interface for expansion cards, so that any card that conforms to it should be compatible with any computer that supports it. Hence expansion cards designed to be compliant with the PCI specification are generally hardware compatible with Power Macintosh computers and with other computers that comply with PCI, including computers that do not use Mac OS. The PCI specification is listed under "Supplementary Documents," in the preface.

Buses conforming to the PCI standard include the following main features:

- operation independent of any particular microprocessor design
- 32-bit standard bus width with a compatible 64-bit upgrade path
- either 5 V or 3.3 V signal levels
- bus clock rate up to 66 MHz
- up to 132 MB per second transfer rate over the 32-bit bus

A PCI bus is typically connected to the computer's processor and RAM system by an ASIC chip called a **PCI bridge.** Power Macintosh computers contain a proprietary bridge chip to connect their PCI buses to the PowerPC processor bus.

Benefits of PCI

PCI represents a needed standard in the desktop computer industry. Because the PCI bus uses the same architecture and protocols to communicate with I/O chips and with plug-in expansion cards, it reduces the cost and complexity of computer hardware. It lets CPU manufacturers provide expandability at minimum cost.

The establishment of the PCI bus standard has benefits for developers of peripheral equipment, too. These benefits include

- delivering a high level of bus performance, enough for most current I/O needs
- letting peripheral equipment developers produce expansion cards that can operate with both Macintosh computers and computers that use other operating systems
- encouraging the large-scale marketing of chips compatible with PCI, which tends to reduce the component cost of peripheral equipment
- providing a relatively simple method for automatically configuring external devices into the user's system during system startup

PCI and NuBus

This section provides some background about the differences between the PCI and NuBus architectures. The PCI bus exhibits a number of fundamental differences from NuBus™, the previous Macintosh bus standard. The most important of these differences are listed in Table 1-1.

Table 1-1 Comparison of NuBus and the PCI bus

Feature	NuBus	PCI bus
Bus clock rate	10 MHz	33 MHz, 66 MHz
Addressing	Geographic	Dynamic
Signal loading	No enforced rules	One load per signal
Transaction length determination	Determined at start of transaction	Determined at end of transaction
Bus termination	Resistor network	Not required
Bus control arbitration	Distributed	Centralized
Addressing spaces	Memory only	Memory, I/O, and configuration
Wait-state generators	Slave only	Slave and master
Kinds of expansion	Cards only	Cards and ASIC chips

Table 1-1 Comparison of NuBus and the PCI bus (continued)

Feature	NuBus	PCI bus
Timeout	255 bus clocks	5 bus clocks
Burst capability	8, 16, 32, or 64 bytes	Any number of bytes
Power allocation	15 W per card	7.5, 15, or 25 W per card

The Macintosh Implementation of PCI

To achieve maximum compatibility with PCI-compliant devices and plug-in cards, the current line of Power Macintosh computers are designed to comply with the *PCI Local Bus Specification,* Revision 2.1. This support includes, as a minimum, the following general areas:

- signal types and pin assignments
- bus protocols, including arbitration
- signal electrical characteristics and timing
- configuration data and card expansion ROM formats
- plug-in card mechanical specifications

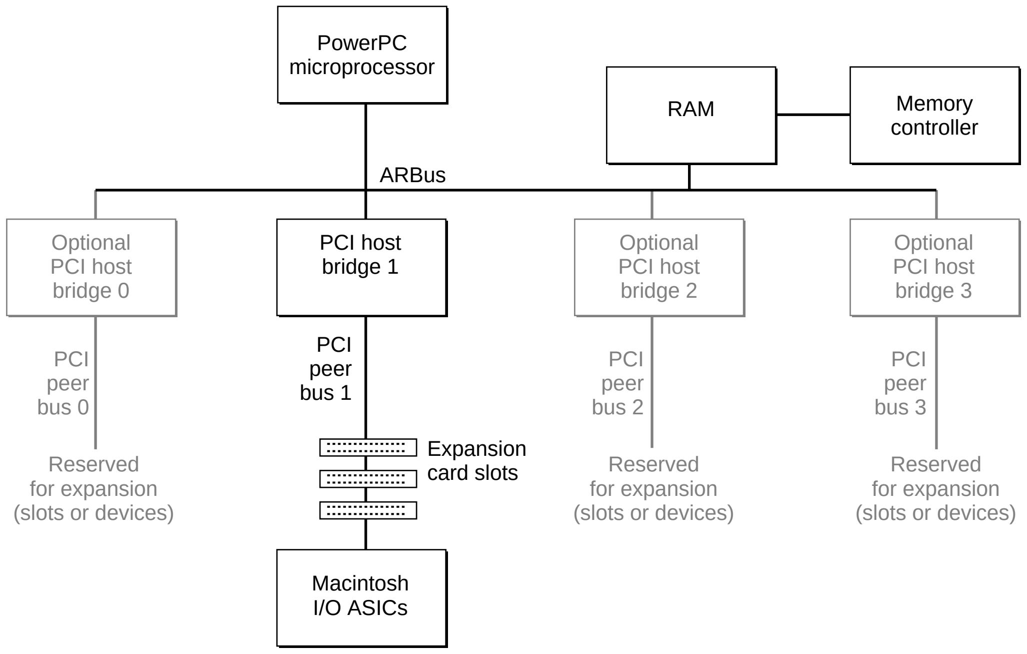
As explained in "Address Allocations" (page 58), a Power Macintosh computer may contain as many as four separate PCI buses for expansion cards, although initial models contain fewer than four.

The next sections contain clarifications and interpretations of the PCI specification that more fully specify the Macintosh implementation of PCI for expansion cards.

Power Macintosh PCI System Architecture

The initial implementation of the PCI bus on Power Macintosh computers supports up to four peer PCI bridge connections to the main processor bus. Figure 1-1 presents a general block diagram of the Power Macintosh system architecture with the PCI bus.

Figure 1-1 PCI system architecture for Power Macintosh


PowerPC microprocessor
RAM
Memory controller
ARBus
Optional PCI host bridge 0
PCI host bridge 1
Optional PCI host bridge 2
Optional PCI host bridge 3
PCI peer bus 0
PCI peer bus 1
PCI peer bus 2
PCI peer bus 3
Expansion card slots
Reserved for expansion (slots or devices)
Reserved for expansion (slots or devices)
Reserved for expansion (slots or devices)
Macintosh I/O ASICs

The ARBus (Apple RISC bus) shown in Figure 1-1 is Apple's implementation of the PowerPC processor bus for Power Macintosh computers. Also note that to date no more than two built-in PCI buses have been included in shipping Macintosh configuration.

PCI Bus Characteristics

The PCI bus on Power Macintosh follows the requirements of the PCI specification described on (page 28). However, the PCI specification allows certain options. Table 1-2 shows the specification options chosen for the first implementation of the PCI bus in Power Macintosh computers.

Table 1-2 PCI options chosen for Power Macintosh

Option	Power Macintosh implementation
PCI clock rate	33 MHz (30 ns cycle time) and 66 MHz
Address/data bus width	32 bits and 64 bits
Signal voltage	5 V, 3.3 V
PCI address spaces supported	Memory,[1] I/O, and configuration
Minimum power supplied	5 V rail: 3 A (15 W) per slot[2] 3.3 V rail: 2 A (6.6 W) per slot[2]
PCI bus arbitration	Fair, round-robin, all slots master-capable
Mechanical bracket	ISA style
Plug-in card expansion ROM	Highly recommended[3]
IDSEL signals	Provided by resistive connections to AD lines
Interrupt routing	INTA#, INTB#, INTC#, INTD# wires combined by OR per slot to provide a unique slot interrupt for each card
LOCK#	Not used by the Macintosh system[4]
PERR#, SERR#	Not used by the Macintosh system
SBO#, SDONE	Not used by the Macintosh system. No cache coherency (snooping) across the PCI bus
JTAG	Not used by the Macintosh system

Notes

[1] The Power Macintosh implementation does not support devices that address memory space below 1 MB.

[2] The PCI specification allocates power per slot, but the Macintosh implementation contains one power allocation for all slots. For example, a three-slot Power Macintosh computer has 9 A of 5 V power or 6 A of 3.3 V power available for PCI cards, which can be installed in any combination among the slots. Apple recommends that cards stay within the proportional allotment: 3 A for 5 V and 2 A for 3.3 V cards. However, configurations with

fewer cards or lower-power cards can support other cards that need more power. These figures are minimum power allocations; some Power Macintosh models may provide more power for PCI cards.

[3] While expansion ROMs are optional in the PCI specification, Apple strongly recommends their inclusion on plug-in cards. True “plug-and-play” operation (plug it in, turn it on, it is available during system boot) can be provided only when an expansion ROM contains both startup firmware and run-time driver code that supports the Open Firmware model for PCI cards. See Chapter 4, “Startup and System Configuration,” for more information on expansion ROM benefits, contents, and data formats.

[4] LOCK# is an optional pin in the PCI specification.

Semaphores must be maintained in main system memory through processor control, using the routines described in “Atomic Memory Operations” beginning on page 426. Power Macintosh does not support the use of semaphores in PCI memory space.

PCI Topology

The Power Macintosh PCI implementation supports a PCI subsystem with the following general restrictions:

- Not more than one PCI-to-ISA bridge can be implemented.
- In systems with two host bridges, ISA bus DMA masters located behind a PCI-to-ISA bridge may target only main memory for DMA transactions, not PCI space.
- In systems with two host bridges, PCI masters located behind one host bridge may not access PCI locations that are mapped behind a PCI-to-PCI bridge located behind the second host bridge.

PCI Host Bridge Operation

The most basic function of the PCI host bridge is to translate between PowerPC processor bus cycles and PCI bus cycles. The bridge in the first implementation of PCI on Power Macintosh provides the following features:

- It supports asynchronous clock operation up to 50 MHz on the PowerPC bus and up to 33 MHz on the PCI bus. The system architecture in Macintosh

PowerPC G3 computers supports asynchronous clock operation up to 100 MHz on the PowerPC bus and up to 66 MHz on the PCI bus.

- It supports split-transaction PowerPC bus implementations.
- It provides dual alternating 32-byte data transaction buffers, one set for bus master transactions initiated by the PowerPC processor bus and one set for bus master transactions initiated by the PCI bus.
- The PowerPC bus can be used in big-endian or little-endian modes. PCI data is always little-endian, and is correctly translated by the PCI host bridge to and from the PowerPC bus in conformance to the PowerPC mode setting. Mac OS is big-endian, so the PowerPC mode setting is big-endian while running Mac OS. For information on translating big-endian and little-endian data formats, see “Addressing Modes” (page 60).
- It supports concurrent PowerPC bus and PCI bus activity.
- Posted writes are always enabled from both PowerPC and PCI masters.
- It supports a 32-byte cache line size.
- It supports and optimizes for the cycle types memory read line and memory write and invalidate. The bridge also accepts memory read multiple cycles from PCI masters and treats them the same as memory read line cycles.
- The longest burst generated as a master or accepted before disconnecting as a target is 32 bytes, the Power Macintosh cache line size.
- It uses medium device select (DEVSEL) timing when operating as a PCI target.

Table 1-3 lists the commands that the Macintosh PCI host bridge supports for all PCI cycle types (all encodings of lines C/BE#[3:0]). The third and fourth columns show whether the bridge can generate the cycle on the PCI bus as a master and whether it can respond to the cycle as a target.

Table 1-3 Bridge support for PCI cycle types

Lines C/BE#[3:0]	Command	Supported as PCI master	Supported as PCI target
0000 (0x0)	Interrupt acknowledge	Yes	No
0001 (0x1)	Special cycle	Yes	No
0010 (0x2)	I/O read	Yes	No
0011 (0x3)	I/O write	Yes	No
0100 (0x4)	Reserved	n.a.	n.a.
0101 (0x5)	Reserved	n.a.	n.a.
0110 (0x6)	Memory read	Yes	Yes
0111 (0x7)	Memory write	Yes	Yes
1000 (0x8)	Reserved	n.a.	n.a.
1001 (0x9)	Reserved	n.a.	n.a.
1010 (0xA)	Configuration read	Yes	Yes
1011 (0xB)	Configuration write	Yes	Yes
1100 (0xC)	Memory read multiple	No	Yes
1101 (0xD)	Dual address cycle	No	No
1110 (0xE)	Memory read line	Yes	Yes
1111 (0xF)	Memory write and invalidate	Yes	Yes

PCI memory space is supported through the bridge transparently—it requires no software abstraction layer to provide functionality. Because the PCI specification defines cycle types that are not directly supported by the PowerPC processor, the Macintosh PCI host bridge provides means to create I/O, configuration, interrupt acknowledge, and special cycles. The bridge generates these cycles in response to the system interface routines described in "PCI Nonmemory Space Cycle Generation" (page 453). To ensure compatibility with future Power Macintosh computers, software must use these routines to access PCI spaces other than PCI memory space.

I/O Space

The PCI Specification requires a 16-bit minimum size I/O space. The first implementation of the PCI bus for Power Macintosh provides a 23-bit I/O space, although the Macintosh address allocation software tries to fit all I/O address space requests within the 16-bit minimum size. The interface to I/O space uses a memory-mapped section in each PCI host bridge's control space. The system determines which PCI host bridge and bridge area to use when accessing each specific card.

Note
In the first PCI implementation for Power Macintosh computers, the bridge posts all PCI write transactions. If the target is in PCI memory space, the bridge writes data directly; otherwise, the bridge generates the necessary I/O, configuration, or special cycle to provide write access. The bridge acknowledges cycle completion even though the transaction may not have been completed at its destination. To check for final write completion, a driver may request a read transaction for the destination device. Verifying that the read transaction has finished will establish that the previous write cycle was flushed from the bridge, without the need to compare data. ◆

Because PCI allocations in I/O space are highly fragmented, high-performance interfaces should try to use the PCI memory space instead of I/O space. The programming interface for I/O cycles is described in "Fast I/O Space Cycle Generation" (page 454).

Configuration Space

The PCI host bridge generates configuration cycles in an indirect manner, similar to mechanism #1 suggested in the PCI specification, using configuration address and configuration data registers to create a single configuration cycle on the PCI bus. The system determines which PCI host bridge and bridge area to use when accessing each specific card. Because configuration cycles must go through a programming interface, high performance interfaces should try to use the PCI memory space instead of configuration space. The programming interface for configuration cycles is described in “Configuration Space Cycle Generation” (page 460).

Interrupt Acknowledge Cycles

Mac OS does not use interrupt acknowledge cycles, but the Macintosh software supports their generation in case some PCI bus chips require them. If a driver needs interrupt acknowledge transactions to control its PCI device, it can use a programming interface that invokes an interrupt acknowledge (read) cycle on the PCI bus. The data returned will be the device’s response, traditionally an Intel-style interrupt vector number. The programming interface for interrupt acknowledge cycles is described in “Interrupt Acknowledge Cycle Generation” (page 466).

Special Cycles

Special cycles are generated by using a programming interface that causes a special cycle (write) on the PCI bus. The special cycle transmits the data message passed to the interface. The programming interface for special cycles is described in “Special Cycle Generation” (page 468).

Maximizing PCI Bus Performance

The guidelines in this section examine the PCI bus commands, the operation of the PowerPC processor to PCI interface bridge chip, achievable PCI bandwidth on PCI Power Macintosh computers, and finally, Mac OS services available to maximize PCI bandwidth.

A good place to start addressing PCI performance on Power Macintosh CPUs is the PCI standard itself. The PCI Bus Specification features a 32-bit data path — upgradeable to 64-bits — with synchronous bus operation up to 33 Mhz, and the ability to transfer a data object on the raising edge of each PCI clock cycle. Assuming that neither the initiator nor the target inserts wait states during each data phase, the maximum theoretical bandwidth over a 32-bit bus is 132 Mbytes/second. This also assumes continuous bursting with a 32-bit data object transferred on each PCI clock cycle. (Apple’s implementation incorporates a 32-bit data bus.)

Because the IB chip competes for system memory along with other system devices, continuous PCI bursting is not possible. Therefore, the achievable PCI bandwidth on Power Macintosh computers is less than the PCI theoretical maximum. Also, the bandwidth is dependent on the PCI target's hardware design and the architecture of the driver software.

A PCI burst transfer is defined by one PCI bus transaction with a signal address phase followed by two or more data phases. One may ask, how can the bus master transfer a data object on each PCI clock cycle? To initiate a bus transaction, the PCI master only has to arbitrate for ownership of the bus one time. The master then issues the start address and transaction type during the address phase. It is the responsibility of the target device to latch the start address into an address counter and increment the addressing from data phase to data phase. (A single-beat read or write transaction is defined by a signal address phase followed by only one data phase.)

For data to be transferred between the PowerPC processor and the PCI target, or for the PCI target to transfer data between system memory, one of the commands shown in Table 1-4 is initiated.

Table 1-4 Commands between PowerPC processor and PCI bus

PCI command	Initiator
I/O Read	Processor generated
I/O Write	Processor generated
Configuration Read	Processor generated
Configuration Write	Processor generated
Memory Read	Processor or PCI Master generated
Memory Read Line	Processor or PCI Master generated
Memory Read Multiple	Processor or PCI Master generated
Memory Write	Processor or PCI Master generated
Memory Write and Invalidate	Processor or PCI Master generated

The I/O Read and I/O Write commands are used to transfer data between the PowerPC processor and the target's I/O space.

The Configuration Read and Configuration Write commands are used to transfer data between the PowerPC processor and the PCI target’s configuration registers during system initialization.

The Memory Read and Memory Write commands are used to transfer data between the PCI Master and the target’s memory space.

The Memory Read Line command is used by the PCI Master to transfer a cache line of data from the PCI target’s memory space.

The Memory Read Multiple command is used by the PCI Master to transfer more than one cache line of data from the PCI Target's memory space.

The Memory Write and Invalidate command is used by the PCI Master to transfer one or more complete cache lines of data to the PCI target's memory space.

A cache line is 32-bytes for Apple Power Macintosh computers.

PowerPC Processor and PCI Commands

The PowerPC processor has a 64-bit data bus and its system memory space defaults to write back cache mode, while the PCI bus is 32-bits wide and the PowerPC processor sets PCI address space to cache inhibit mode. For PowerPC initiated read and write transactions between PCI memory space, the IB chip (the PowerPC Processor to PCI Bridge) will initiate basically one of the three following types of PCI commands:

1. a single-beat Memory Read or Write command;
2. a Memory Read or Write command with two data phases — defined as a burst transaction;
3. a Memory Read Line or Memory Write and Invalidate command that bursts a 32-byte cache line.

IMPORTANT

The PPC processor will not burst to or from address space marked cache inhibited. Therefore, under default cache settings, the IB chip will not initiate the Memory Read Line or Memory Write and Invalidate commands to a PCI target.

As per the PCI Specification, PCI Power Macintosh Computers support PCI I/O space. PCI I/O commands and Mac OS services available for them are addressed later in this Technote.

Bursting from PowerPC to PCI

Provided software is written to utilize floating-point load and store instructions, as opposed to integer operations, the IB chip will burst a two-beat Memory Read or Memory Write command (two 4-byte data phases with one PCI transaction). The PowerPC floating-point data is 8-bytes wide and integer data is 4-bytes. Utilizing floating-point instructions in effect nearly doubles the PCI bandwidth over single-beat PCI Memory Read or Write commands. This is worth investigating for solutions where the PCI hardware does not support cache line bursting.

If the PCI target's address space is set to write through cache mode, the IB chip will perform an eight-beat burst read on PCI with the Memory Read Line command. This translates to a cache line, eight 4-byte long words, i.e. 32-bytes.

If the PCI target's address space is set to write back cache mode, the IB chip will perform an eight-beat burst write on PCI with the Memory Write and Invalidate command.

IMPORTANT

Extreme care must be taken for burst writes to PCI address space to perform appropriate cache flushing.

Bursting from PCI to PowerPC

If the address is aligned on an 8-byte boundary, the IB chip will respond to PCI Memory Read and Memory Write commands by a two-beat PCI transaction to align two 32-bit PCI data words to the 64-bit PowerPC bus. On non-8-byte-aligned addresses, single-beat transactions are implemented.

The PCI Memory Write and Invalidate command will perform an 8-beat transaction if the address is aligned on a 32-byte boundary.

The PCI Memory Read Line or Memory Read Multiple commands perform an eight-beat transaction if the address is aligned to an address less than or equal to 8-bytes less than the next 32-byte boundary. The PCI Memory Read Line and Memory Read Multiple commands are treated the same by the IB chip, in either command case the IB chip disconnects after an eight-beat transaction, which is one 32-byte cache line.

Note
Keep in mind that the main memory space is set to write back cache mode.

As mentioned earlier, 132 Mbytes/sec is the maximum theoretical bandwidth across a 32-bit PCI bus at 33 Mhz. Table 1-5 and Table 1-6 show the maximum achievable bandwidth that can be expected, depending on the type of PCI transaction performed. The values shown are not guaranteed, but are realistic ranges that have been measured moving large buffers (many thousands of bytes) to average out PCI arbitration PCI wait states across a Power Macintosh Computer's PCI bus.

The bandwidth performance numbers shown in Table 1-5 and Table 1-6 are based on the following assumptions:

Bus speed:

- Processor Bus is running at minimally 40 Mhz
- PCI Bus is running at 33 Mhz

PCI target responses during PowerPC processor to PCI transactions:

- PCI targets are medium DevSel_ timing with NO inserted wait states for reads and writes.
- PCI target does not assert Stop_ to disconnect bus.

PCI master requirements during PCI master with system memory transactions:

- PCI master is able to source data within one clock of Frame_ assertion with no inserted wait states for subsequent data phases.
- PCI master is able to sink data with no inserted wait states for subsequent data phases once the host bridge asserts Trdy_.

- PCI master is able to start its next transaction within two clocks from the PCI bus returning to the Idle state from its previous transaction.

Table 1-5 PowerPC processor to PCI maximum bandwidth summary

Bus master	**Transaction description**	**Bytes per transaction**	**PCI bandwidth, MB/s**	**PowerPC setup**
Processor	Write To PCI	4	20	Integer Store
Processor	Write To PCI	8	40	Floating Point Store
Processor	Write To PCI	32	85	PCI Copyback
Processor	Read from PCI	4	11	Integer Load
Processor	Read from PCI	8	20	Floating point Load
Processor	Read from PCI	32	40	PCI WriteThru

Table 1-6 shows the PCI master to system memory bandwidth measurements for a 33 MHz PCI bus in a system with a 40 MHz processor bus.

Table 1-6 PCI master to system memory maximum bandwidth summary

PCI master	**Transaction description**	**Bytes per transaction**	**PCI bandwidth, MB/s**	**PowerPC command**
PCI master	Write to memory	4	20	Mem Write
PCI master	Write to memory	8	35	Mem Write
PCI master	Write to memory	32	80	Mem Write and Invalidate

Table 1-6 PCI master to system memory maximum bandwidth summary

PCI master	Transaction description	Bytes per transaction	PCI bandwidth, MB/s	PowerPC command
PCI master	Read from memory	4	10	Mem Read
PCI master	Read from memory	8	15	Mem Read
PCI master	Read from memory	32	30	Mem Read line multiple

For number of bytes per transaction 4 indicates single-beat; 8 equals two-beats; and 32 is an 8-beat transaction.

Mac OS & Services That Maximize PCI Throughput

This section discusses the Mac OS services available to maximize PCI throughput.

Beginning with the Mac OS version 7.5.2 release, a DSL (Driver Services Library) that implements all programming interface services is available for drivers. The complete API for the DSL is documented in Chapter 11, "Driver Services Library."

To coordinate I/O operations that transfer buffers between system memory and PCI address space, the Macintosh OS provides two functions with the DSL: `PrepareMemoryForIO`, and `CheckpointIO`. The `PrepareMemoryForIO` function allocates resident system memory to buffers, provides logical and physical address information, and in conjunction with `CheckpointIO` manages coherency between system memory and the PowerPC caches. `CheckpointIO` is called after the buffer transfer is complete and either relinquishes the memory back to the OS and adjusts the processor caches for coherency, or prepares for another IO transfer.

Note
`PrepareMemoryForIO` should not be confused with PCI I/O space. It is for buffers whether they are located in PCI memory or PCI I/O space. ◆

PrepareMemoryForIO is an example of a service in the DSL; PCI cards that have DMA hardware should use PrepareMemoryForIO to locate physical addresses in system memory. Older I/O expansion cards would typically use the toolbox call GetPhysical to locate physical addresses in system memory. To be fully compatible with the present and future Mac OS releases, drivers should only use the DSL services described in Chapter 11, "Driver Services Library."

Remembering that PCI address space defaults to cache inhibit mode, to enable the PowerPC to burst to areas of PCI memory space, that area must be set to cacheable. This can be done with the SetProcessorCacheMode function described in "SetProcessorCacheMode" (page 372). Set the desired PCI address space to kProcessorCacheModeCopyBack for cache line writes and kProcessorCacheModeWriteThrough for cache line reads.

IMPORTANT

Extreme care must be taken for burst writes to PCI address space to perform appropriate cache flushing. ▲

Be advised that the SetProcessorCacheMode has an undocumented limitation. The PowerPC address space is divided into sixteen 256-Mbyte segments that are distinguished by the upper 4-bits of the effective address. The SetProcessorCacheMode is only capable of changing the cache setting for one contiguous section of memory per 256-Mbyte segment. Therefore, if two PCI cards are configured where they both have PCI address assignments in the same segment only one card can change its address space cache setting.

For example, if two cards (card x and card y) have addresses mapped into segment 8, one at 0x80800000 and another at 0x80801000, the first call to SetProcessorCacheMode from the driver of card x to make a cacheable address space in segment 8 will work. A second call, say from the driver of card y, to modify the cache setting in segment 8 will not work nor will it report an error. This scenario will most likely result in a lower than expected performance for card y, because card y address space is actually cache inhibited which disables PCI transactions of 32-byte cache lines. If the two cards are mapped into different segments, such as 8 and A, then they both can modify the cache settings within their perspective segments. This limitation will be relaxed in the future.

Extensions to the BlockMove routine have been incorporated in the DSL that optimize performance on the PowerPC CPU family. In particular, BlockMoveData has been optimized for data that is cacheable and BlockMoveDataUncached for data that is cache inhibited. The difference between the cached and uncached versions of these instructions is that, for BlockMoveData, the PPC dcbz instruction is used to avoid the logically unnecessary read of the destination cache blocks. `BlockMoveDataUncached` does not use the dcbz instruction because dcbz is extremely slow for address space marked cache inhibited or cache write thru.

The difference between `BlockMove` and `BlockMoveData` versions is whether or not the block being moved contains 68K instructions. If the data does contain 68K instructions `BlockMove` must be called which also flushes the DR (Dynamic Recompilation) Emulator's cache. This is costly time-wise, so if the block does not contain 68K instructions, be sure to use `BlockMoveData` or `BlockMoveDataUncached`. Also with performance in mind, when appropriate the BlockMove routines will align the source and destination address to utilize floating-point load and store instructions.

For transfers of large buffers between PCI cards the `BlockMoveData` or `BlockMoveDataUncached` functions should be used, depending if the destination address space is marked write back cacheable or not. Native PCI drivers most likely will not need to consider the non-Data variant of the `BlockMove` routines because destination buffers either in PCI address space or system memory will probably not need to execute 68K code.

To initiate a PCI burst of a cache line, use the `BlockMoveData` function. Provided the PCI address space is marked cacheable as explained earlier, the `BockMoveData` function forces the IB chip to burst 32-byte cache lines — eight-beat data phases per PCI command transaction.

To read or write PCI I/O space, the Expansion Bus Manager provides routines to transfer data — byte, word, or long word (8, 16, or 32 bits, respectively) — using PCI I/O Read and I/O Write commands. The Expansion Bus Manager is part of the ROM firmware in PCI Power Macintosh CPUs. These routines also perform appropriate byte swapping. For a further description, refer to Chapter 12, "Expansion Bus Manager.". PCI cards that are limited to I/O space, and do not incorporate PCI memory space, are limited to PCI I/O Read and I/O Write commands to transfer data between the PowerPC processor and PCI target. If PCI I/O data needs to be processed quickly, note there is a significant performance hit using Expansion Manager Routines. These routines are intended for PCI targets that have I/O registers or low bandwidth I/O buffers. The IB chip does not burst PCI I/O Read nor burst PCI I/O Write commands.

As described in "Fast I/O Space Cycle Generation" (page 454), the PCI property `assigned-addresses` provides vector entries that represent physical addresses on PCI cards. Using the `APPL,address` property, a driver can locate a logical address of a physical I/O resource. By accessing the logical I/O address, the IB chip generates the appropriate PCI I/O command. Therefore a driver can generate PCI I/O commands without using the Expansion Bus Manager Routines; the same way it accesses PCI memory space. This provides the fastest way to access I/O space, but note it does not perform the byte swapping provided by the Expansion Bus Manager routines.

Also note, the Expansion Bus Manager provides OS services to generate PCI Configuration Read, Configuration Write, Interrupt Acknowledge, and Special Cycle commands.

To maximize bus performance, utilize the services available in the Driver Services Library, and pay close attention to PCI chip selection, in particular, chips that can execute cache line burst transactions with Memory Read Line, Memory Read Multiple, and Memory Write and Invalidate commands.

To maximize your PCI card's performance on the Power Macintosh platform. As a PCI target, your card should

- minimize the number of wait states
- accept burst transactions of cache line size without disconnecting
- support 8-byte burst transactions if it cannot support cache line size burst transactions

As a PCI master, your card should

- minimize the number of wait states for transactions and arbitration
- support linear burst ordering and be able to read or write at least one whole cache line of data
- support the memory read line or memory read multiple cycle types for read transactions
- support the memory write and invalidate cycle type for write transactions

PCI Transaction Error Responses

The PCI host bridge responds to system error and exception conditions in a manner that prevents the system from hanging. The bridge tries to signal the error or exception and terminate the transaction gracefully. Buffers are made available for use after the exception or error. Error translations when the PCI host bridge acts as a PCI master (that is, as an agent for the PowerPC bus master) are shown in Table 1-7.

Table 1-7 Bridge master errors

Transaction	PCI target response	Result
Write	No DEVSEL (master abort)	Data discarded after posting. Received master abort error interrupt generated.
Write	Target abort	Data discarded after posting. Received target abort error interrupt generated.
Read	No DEVSEL (master abort)	Machine check exception (bus error) generated. Received master abort error interrupt generated.
Read	Target abort	Machine check exception (bus error) generated. Received target abort error interrupt generated.

Error translations when the PCI host bridge acts as a PCI target (that is, as an agent for the PowerPC bus target) are shown in Table 1-8.

Table 1-8 Bridge target errors

Transaction	PowerPC bus target response	Result
Write	Bus error	Data discarded after posting. Signaled target abort error interrupt generated (though target abort is not signaled because the write was already posted).
Read	Bus error	Generate target abort. Signaled target abort error interrupt generated.

PCI Card Characteristics

Every PCI expansion card should contain code in its expansion ROM conforming to IEEE Standard 1275. Among other tasks, this code helps build a configuration structure called a *device tree.* The requirements for this code (and the benefits of its inclusion in expansion ROMs) are discussed in "The Open Firmware Startup Process" (page 83).

▲ **WARNING**
Expansion cards should follow the mechanical specifications given in *PCI Local Bus Specification,* Revision 2.0, exactly. In particular, short PCI cards for Macintosh computers should not be longer than the 6.875-inch (174.63 mm) dimension specified. In some Macintosh models, 6.875 inches represents the maximum length for a PCI card, while in other models cards may be any length up to 12.283 inches. ▲

PCI Video and Display Card Characteristics

Frame buffers in PCI video cards must support the existing Macintosh big-endian pixel ordering. If accessible in more than one data format, frame buffers on cards should also support multiple views (called *apertures*) by being mapped in different formats to separate areas of memory. These concepts are described in "Frame Buffers" (page 64).

PCI video display cards in Power Macintosh computers should define certain properties in the device tree to let the cards function during system startup. These properties are discussed in Chapter 5, "PCI Open Firmware Drivers."

PCI video display devices should provide an interrupt to mark vertical blanking intervals. Mac OS utilizes this interrupt to do cursor and screen updates to avoid flicker. If the hardware interrupt for vertical blanking is not provided, a time management task may be installed. For more information on this subject, see Chapter 13, "Graphics Drivers."

Power Macintosh computers support the ISA bracket for PCI expansion cards.

Hard Decoding Device Address Space

Hard decoding is a practice in which a PCI device does not employ the fully relocatable PCI base address method for defining its address spaces. Instead, it chooses an address space and decodes accesses to it, with no indication to the system that it has done so.

While hard decoding is not recommended by the PCI specification, certain designs based on Intel microprocessor architecture have used it—for example, VGA and IDE expansion cards. Hard decoding cripples the ability of system software to resolve address conflicts between devices. A problem exists when multiple devices that hard decode the same address space are plugged into a system, or when a device does not notify the system that it has hard decoded portions of the address space. If the system knows the range of addresses that a device hard decodes, addresses can be assigned to fully relocatable devices around the spaces already taken. However, if two devices that hard decode the same space are installed in the system, address conflicts can be resolved only by the system turning off one of the devices.

You can never hard decode addresses below 1 MB (for example, VGA addresses A0000 through BFFFF) because the Power Macintosh implementation of PCI does not support devices that address this space. Moreover, it is very common for a user to plug in multiple display cards to use multiple monitors. If more than one of these cards hard decodes the VGA addresses, only one will be enabled, and it cannot be guaranteed which device that will be. It is essential, therefore, that devices which hard decode address spaces after reset provide a method to turn off their hard-decoding logic. The result of turning off hard decoding must mean that the device responds to accesses only in the address spaces that are assigned to it through the PCI base register interface. This method can be executed in FCode during startup, before the device enters its `reg` property into the device tree. See Chapter 4, “Startup and System Configuration,” for more details.

To summarize, avoid hard decoding to ensure that your card will always be allocated address space. If a device cannot turn off hard decoding, its FCode must enter a fixed address `reg` property entry into the device tree.

Nonvolatile RAM

Power Macintosh computers that support the PCI bus contain nonvolatile RAM (NVRAM) chips with a minimum capacity of 4 KB. A typical allocation of NVRAM space is described in "Typical NVRAM Structure" (page 443).

An important use of the Power Macintosh NVRAM is to store the `little-endian?` variable, discussed in "Addressing Mode Determination" (page 63).

In Macintosh computers that support the New World architecture, hard-coded offsets to locations in NVRAM for Mac OS PRAM variables are no longer used in the same way. Instead, NVRAM is divided into variable-sized partitions that are available to the Mac OS, Open Firmware, or any other client. This partitioning scheme is based on the Common Hardware Reference Platform (CHRP) specification.

Mac OS PRAM variables reside in the Mac OS partition in NVRAM, and API calls to modify PRAM refer to offsets within the Mac OS partition. The Mac OS also stores device configuration properties in NVRAM that are used by Open Firmware at startup time. These properties are stored in an Open Firmware config variable. The Mac OS uses Name Registry functions to load this information into NVRAM to provide device boot-time configuration services for Open Firmware drivers and Mac OS drivers during subsequent startup of the system.

See, Chapter 3, "Introduction to the NewWorld Architecture," for a discussion of the significant software changes implemented in the New World architecture.

C H A P T E R 2

Data Formats and Memory Usage

This chapter describes the memory allocations that Power Macintosh computers reserve for PCI use and defines the data formats used with PCI buses. It discusses PCI bus cycles, big-endian and little-endian addressing modes, and the storage of data in frame buffers.

Address Allocations

The first implementation of Power Macintosh computers that uses the PCI bus reserves specific areas of the overall 32-bit address space for use by PCI expansion cards. Address allocation in the first Macintosh PCI system follows these general principles:

- A Power Macintosh system may contain up to four peer PowerPC–to–PCI host bridges. The functions of these bridges are described in "PCI Host Bridge Operation" (page 39).
- After each PCI host bridge, PCI-to-PCI bridges may be added in any configuration to create up to 256 PCI buses in the Power Macintosh hardware, the maximum that the PCI specification allows. However, properties that must be stored on disk or in NVRAM between system startups can be addressed only to five levels of PCI-to-PCI bridges behind each host bridge. Therefore the number of hardware PCI buses that the system software supports fully is limited to six times the number of host bridges, or 24 buses maximum.
- More than 1.8 GB of address space is allocated for PCI memory space.
- Remaining regions of the Macintosh 32-bit address space are allocated to system RAM, ROM, and control.

The general memory allocation scheme for the first implementation of Power Macintosh computers with PCI buses is shown in Table 2-1.

IMPORTANT

The information in Table 2-1 is for illustrative purposes only. Neither hardware nor software should rely on the address map described therein.

Table 2-1 Power Macintosh memory allocations

Address range	Usage
0x0000 0000–0x7FFF FFFF	System RAM
0x8000 0000–0xEFFF FFFF	Available to PCI expansion cards
0xF000 0000–0xF1FF FFFF	PCI host bridge 0 control
0xF200 0000–0xF3FF FFFF	PCI host bridge 1 control
0xF400 0000–0xF5FF FFFF	PCI host bridge 2 control
0xF600 0000–0xF7FF FFFF	PCI host bridge 3 control
0xF800 0000–0xF8FF FFFF	System control
0xF900 0000–0xFEFF FFFF	Available to PCI expansion cards
0xFF00 0000–0xFFFF FFFF	System ROM

PCI Bus Cycles

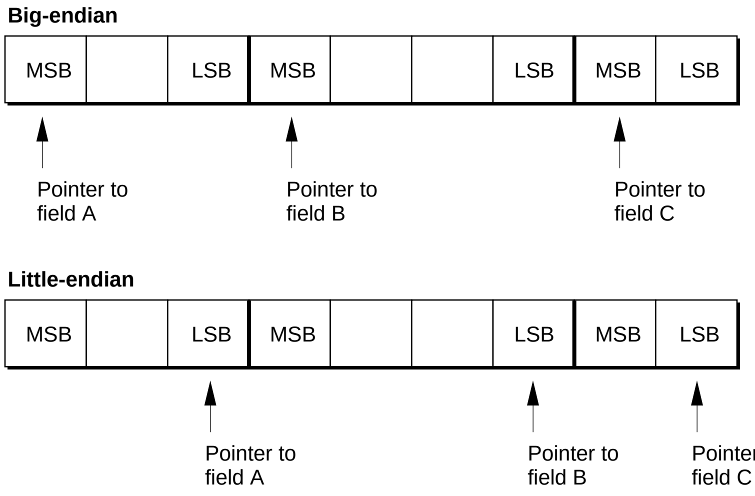
Besides defining cycles for PCI memory space, which is directly addressable by the PowerPC processor, the PCI specification supports four other types of cycles—I/O space, configuration space, interrupt acknowledge, and special—which are not directly supported by the PowerPC architecture. To provide a PCI-compliant interface, Macintosh bridges create these additional address spaces and cycle types by accessing memory-mapped regions of the bridge control space shown in Table 2-1. Because the additional spaces and cycle types are manufactured by the bridge, they are abstracted from driver code and expansion card firmware by the interface routines defined in Chapter 12, “Expansion Bus Manager.” Using these routines, you can create all types of data transactions on Macintosh PCI buses in a hardware-independent way.

Addressing Modes

There are two ways that multibyte data fields may be addressed: **big-endian** addressing, where the address for the field refers to its most significant byte, and **little-endian** addressing, where the address for the field refers to its least significant byte.

These two types of data organization are illustrated in Figure 2-1, which shows a region of memory containing successive fields that are 3, 4, and 2 bytes long. MSB and LSB indicate the most significant and least significant bytes in each field, respectively.

Figure 2-1 Big-endian and little-endian addressing



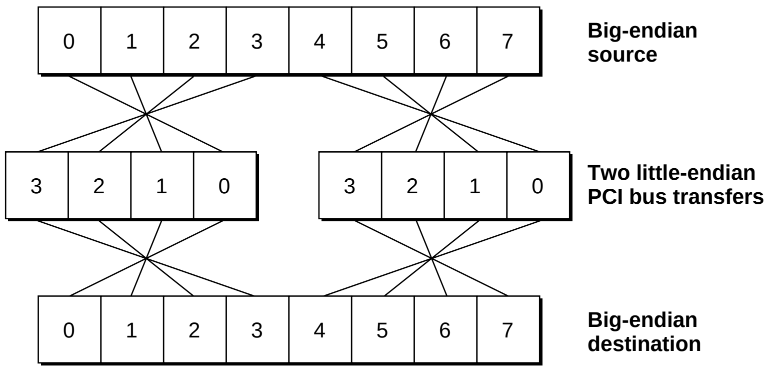
Since data fields are normally stored in RAM by writing from lower to higher addresses, big-endian addressing also means that the field's lowest address in physical memory contains its most significant byte; little-endian addressing means that the field's lowest address contains its least significant byte.

If the Macintosh system always wrote and read multibyte data fields in one operation, it wouldn't matter whether the fields were addressed in big-endian or little-endian mode. For example, if the hardware always transferred an 8-byte field in a single transaction, using 64 bit-lines, it would be immaterial whether the location of the field were defined by referencing its most significant byte or its least significant byte. But when data fields are transferred over buses of limited width, they must often be divided into subfields that fit the capacity of the bus. For a more detailed discussion of endian issues, see Appendix A, "Big-Endian and Little-Endian Addressing."

Addressing Mode Conversion

With the PCI bus (in the 32-bit version that Power Macintosh uses), fields more than 4 bytes long must be transferred in multiple operations. When writing a field from one location to another by means of multiple transfers, the bus must take into account the addressing modes of both the source and destination of the data so that it can disassemble and reassemble the field correctly. One way to convert data from one addressing mode to the other is to reverse the order of bytes within each field, so that a pointer to the most significant byte of a field will point to the least significant byte, and vice versa. Note that the addresses of the data bytes do not change. This technique, called **address-invariant byte swapping,** maintains the address invariants of data bytes. It is illustrated in Figure 2-2.

Figure 2-2 Big-endian to big-endian bus transfer



Note
The difference between big-endian and little-endian formats applies only to data; the Macintosh system always transfers addresses as unbroken 32-bit quantities. ◆

PowerPC processors and processors of the Motorola 68000 family use big-endian addressing; Intel processors and the PCI bus use little-endian addressing. Different I/O chips, expansion card memories, and peripheral devices may use one addressing mode or the other, so data in versatile computing systems such as Power Macintosh must often be accessed in either form.

Figure 2-2 illustrates what happens when data from a big-endian source passes over the little-endian PCI bus and is written to a big-endian destination. The bytes in the source and destination are numbered from 0 to 7.

The Power Macintosh hardware supports both big-endian and little-endian addressing. To accommodate various combinations of source and destination byte formats, Power Macintosh systems contain two mechanisms that translate between these addressing modes:

- A group of byte-reversed indexed load and store actions are included in the PowerPC instruction set—for example, the `lwbrx` (load word byte-reversed index) instruction. These instructions can convert either big-endian or little-endian data to the other format, because the two formats are complementary. C programs can perform the same operations by using endian swap routines.
- The PowerPC processor supports a little-endian addressing mode that changes the way in which real addresses are used to access physical storage. It applies a logical exclusive-OR operation with a constant to the lowest 3 bits of the address, using a different constant for each size of data. This modifies each address to the value it would have if the PowerPC processor used little-endian addressing.

The PowerPC system software also contains a pair of utility routines that convert 16- and 32-bit values into the other endian format by means of byte swapping. These utilities are described in "Byte Swapping Routines" (page 469).

For more detailed information about endian conversion, see Appendix A, "Big-Endian and Little-Endian Addressing."

Programs and subsystems that exchange data only internally can usually adopt either big-endian or little-endian addressing without taking into account the difference between the two. As long as they operate consistently, they will always store and retrieve data correctly. Systems that exchange data with other devices or subsystems, however, including those that communicate over the PCI bus, may need to determine the addressing mode of the external system and adapt their data formats accordingly.

When designing PCI cards for Power Macintosh computers, including their associated software, observe the following general cautions about byte formats:

- The PowerPC microprocessor and the PCI host bridges are set for big-endian addressing when running a big-endian operating system such as Mac OS.
- Most compilers do not provide support for switching data from one addressing mode to another or for using the PowerPC mechanisms that switch modes. Such support can be provided, for example, by a set of C macros that redefine the access mechanisms for basic data types.
- Frame buffers for video and graphics must support the Macintosh big-endian pixel format, as described in “Frame Buffers,” later in this chapter.

Addressing Mode Determination

It is possible to determine whether a system uses big-endian or little-endian addressing by comparing the way it arranges bytes in order of significance with the way it addresses fields. For example, the code shown in Listing 2-1 makes this test.

Listing 2-1 Endian addressing mode test

```
typedef unsigned short  half;
typedef unsigned char   byte;

union {
    half H;
    byte B[2];
    } halfTrick;
halfTrick ht;
ht.H = 0x2223;
if (ht.B[0] == 0x22)
```

```
    printf("I'm big-endian");
else
    printf("I'm little-endian");
```

An important global variable that the Power Macintosh startup firmware stores in nonvolatile RAM is called `little-endian?`. It contains a value of 0 if the last operating system run on the computer used big-endian addressing or –1 if the last operating system used little-endian addressing. Each time the Power Macintosh startup firmware loads an operating system, it checks to see whether the system's big-endian or little-endian operation matches the value in `little-endian?`. If the match fails, the Power Macintosh startup firmware changes the value in `little-endian?` and begins the Open Firmware startup process again. The Power Macintosh nonvolatile RAM is described in "Nonvolatile RAM" (page 56).

Frame Buffers

Frame buffers in PCI video and graphics cards must support the existing ways that Power Macintosh computers handle graphical data, including the storage of pixel information in memory and the presentation of that information in various formats.

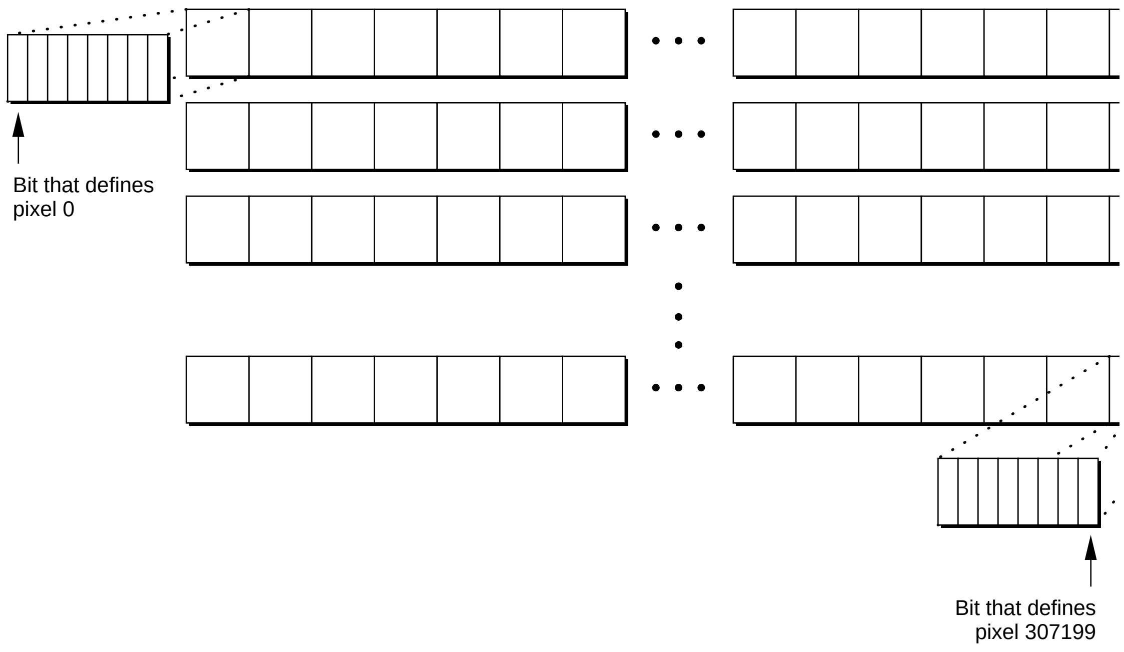
Pixel Storage

The Macintosh pixel storage format is big-endian. This format has the following general characteristics:

- All the bits that define any single pixel on the screen (ranging from 1 to 32 bits) are adjacent in memory.
- The bit groups that define each pixel are successive and contiguous in memory, starting with the pixel at the upper-left corner of the screen and ending with the pixel in the lower-right corner of the screen.

For example, a frame buffer that defines a screen 640 pixels wide by 480 pixels high (307,200 pixels), using 1 bit per pixel, contains 38,400 bytes. The most significant bit of the first byte corresponds to pixel 0, located in the upper-left corner of the screen. The least significant bit of the last byte corresponds to pixel 307199. This example is diagrammed in Figure 2-3.

Figure 2-3 Sample frame buffer format



If the same frame buffer had a color depth of 8 bits (thereby containing 307,200 bytes), all of the first byte would be used to store information about pixel 0 and all of the last byte would be used to store information about pixel 307199.

For further information about Macintosh pixel formats, see Appendix B, “Graphic Memory Formats.”

Note
Data in PCI control, status, and configuration registers for PCI video cards on Power Macintosh computers must be in little-endian format. ◆

Frame Buffer Apertures

In some situations, a frame buffer on a PCI expansion card may need to support data accesses in more than one format. For example, a frame buffer may need to store frame buffer data from a big-endian source in three different formats—RGB, a little-endian source in RGB, and a YUV data format. To provide multiple formats on the fly, a PCI card can create multiple apertures of its frame buffer.

An **aperture** is a logical view of the data in a frame buffer, organized in a specific way. The PCI card converts its frame buffer contents into the required format for each aperture, and maps each aperture into a different range of memory addresses.

Each aperture is defined by specifying its starting address in memory, its width and height in pixels, and the format and size of each pixel description. The aperture definition may also include a *row bytes* value, giving the address offset between successive rows. Although each aperture normally has a different pixel description, the arrangement of pixels in the frame is the same for all apertures; this arrangement starts with the upper-left pixel and proceeds as described in the previous section. An aperture may represent the whole frame buffer or any region within it.

One important use for apertures is to provide both big-endian and little-endian views of a frame buffer. Providing both views can eliminate the need for support of the byte-swapping operations. For example, in a PCI card's memory space of 16 MB, 8 MB could be allocated for a big-endian aperture and registers and 8 MB could be allocated for a little-endian aperture and registers. Mac OS running on the PowerPC processor would access the big-endian aperture, while a frame-grabber PCI master card that supported a little-endian pixel format would access the little-endian aperture.

Apertures are supported by the device drivers associated with a PCI card, which must respond to calls that query and select the card's aperture capabilities. Each aperture can be treated as a virtual device, to be opened and closed separately from other apertures. A driver can treat the physical organization of the frame buffer as an aperture as well, without subjecting it to mapping or format conversion.

For more information on apertures see *PCI Multimedia Design Guide,* published by the PCI SIG. You can contact the PCI SIG at the address given on (page 28).

P A R T   T W O

The Open Firmware Process

This part of *Native Drivers for Power Macintosh Computers* describes the Open Firmware process and tells you how it works with Power Macintosh computers running the Mac OS. It contains three chapters:

- Chapter 3, "Introduction to the NewWorld Architecture," describes how the NewWorld Architecture works from an organizational and execution flow standpoint, and describes differences from older architectures.
- Chapter 4, "Startup and System Configuration," describes how PCI-compatible Macintosh computers recognize and configure peripheral devices.
- Chapter 5, "PCI Open Firmware Drivers," discusses the general technical requirements for Open Firmware drivers for PCI devices—drivers that are used with the Open Firmware startup process.

Part 4 of this book covers native device drivers that support the PCI-bus compatible Power Macintosh architecture. Part 4 begins on (page 137).

Chapter 5, "PCI Open Firmware Drivers," discusses Open Firmware drivers, which control PCI and boot devices during the Open Firmware start-up process.

C H A P T E R 3

Introduction to the NewWorld Architecture

The NewWorld architecture is the basis for Mac OS start-up and system ROM (read only memory) functionality for all new Macintosh computers, beginning with the iMac.

This chapter describes how the NewWorld architecture works from an organizational and execution flow standpoint and describes differences from older architectures. It briefly covers the ROM organization prior to NewWorld as background, then explains the NewWorld architecture and execution flow.

While the focus of the information contained in this chapter is on Mac OS behavior in the NewWorld architecture, the Macintosh on-board bootROM and Mac OS ROM image file components of the NewWorld architecture are operating system independent. The mechanisms behind the software engineering techniques used to support Mac OS in the NewWorld architecture can be applied to other operating systems.

The Macintosh ROM and The NewWorld Architecture

Historically, the Macintosh ROM has been structured as one monolithic chunk of firmware in ROM, containing both low-level and high-level code. That is, it contained the routines needed by the computer at power-up time (hardware knowledge, initialization, diagnostics, drivers, and such), as well as quite a bit of higher level Mac OS code.

While a computer needs to have a ROM with hardware-specific code in order to boot, the higher level code was also included in the Macintosh ROM because the Macintosh ROM had its genesis in the original 128K Macintosh computer back in 1983. In those days, ROM was cheaper than RAM, and the available disk space (which was floppy based) was at a premium. These factors are certainly no longer valid reasons for keeping (permanently locking) everything in ROM. RAM is relatively inexpensive now and it is faster than ROM.

The NewWorld architecture separates the hardware-specific and higher level system software into two logically distinct pieces. In the NewWorld model, one piece, the bootROM, holds most of the hardware-specific components needed to boot the specific logic board implementation, while the other, the Mac OS ROM image file, contains boot-time Mac OS routines and hardware-specific software components that are common to many Macintosh computers.

Low-level hardware-specific code still exists in firmware (ROM) in order to handle the computer's start-up activities. This code fits into one ROM called the bootROM. The bootROM has the hardware-specific code and description of the hardware needed to start up the computer and provide common hardware access services the operating system might require. Open Firmware is in the bootROM and provides access to the basic hardware subsystems and a Forth interpreter. Macintosh computers that support the NewWorld architecture have Open Firmware version 3.0 or later. Open Firmware is defined in Chapter 4, "Startup and System Configuration," and Chapter 5, "PCI Open Firmware Drivers."

The Macintosh Toolbox is no longer in the on-board ROM. Only Open Firmware boot services still exist in the on-board hardware ROM. The Toolbox and other services necessary to boot the operating system after hardware initialization has taken place are in a disk file called "Mac OS ROM." The Mac OS ROM image file (also known as the bootinfo file) is loaded from a mass storage device. The contents of the Mac OS ROM file is decompressed and stitched into the memory map as if it were a firmware ROM (that is, it is write-protected in the memory map). This is where the term ROM-in-RAM comes from.

Differences Introduced by the NewWorld Architecture

From a user and application developer point of view, the NewWorld architecture is implemented to be as compatible as possible with previous Macintosh system architectures. However, there are differences that developers, particularly developers interested in participating in the boot process, should be aware of. The information in the following sections briefly describes the key differences. The boot process is defined in "NewWorld Boot Process" (page 77), and in "Startup Sequence in the NewWorld Architecture" (page 92).

Boot Devices

Devices connected to third-party expansion cards that typically expect to participate in the boot process must provide **Open Firmware boot drivers** in the expansion ROM on the card. These drivers are written in FCode, the forth programming language. Without the FCode driver, devices are not recognized early in the startup process. Expansion cards without FCode are recognized only after the Mac OS has started and the associated runtime drivers are located on the disk. For example, a disk drive attached to a SCSI expansion card that does not include proper FCode in the expansion ROM is not able to boot, but the drive shows up on the desktop in the Finder after the SCSI card runtime driver is loaded from disk by the Mac OS.

Devices connected to Macintosh built-in I/O, such as ATA storage devices and on-board Ethernet ports, are supported by FCode in the Macintosh bootROM on the main logic board. Third-party PCI cards on the PCI bus, such as SCSI mass storage device controllers, Ethernet cards, and graphics display controllers, require Open Firmware FCode drivers in the PCI card expansion ROM in order for the devices connected to those cards to be recognized early in the boot process, before startup control switches from Open Firmware to the Mac OS.

Open Firmware provides boot mechanisms and a description of the system hardware, in the form of a **device tree**. The IEEE Std 1275-1994: Standard for Boot (Initialization, Configuration) Firmware: Core Requirements and Practices documentation, along with associated bindings, provides the specification for Open Firmware. The current bindings are available at

http://playground.sun.com/1275/home.html

or at Apple's mirror site at:

http://bootrom.apple.com

The Macintosh implementation of Open Firmware is discussed in Chapter 4, "Startup and System Configuration." NewWorld startup disk behavior is described in "Startup Disk Control Panel" (page 94). An introduction to Open Firmware drivers and the Open Firmware environment on the Macintosh is provided in "PCI Open Firmware Drivers" (page 111).

RAM

Another major difference introduced with the NewWorld architecture is that logical RAM and physical RAM are no longer mapped one-to-one, as they have been in previous PCI-based Macintosh computers. This means that well-behaved software must call the `LogicalToPhysical` or `PrepareMemoryForIO` functions, as described in "Memory Services Used During I/O Operations" (page 352). Software logic that assumes the logical and physical addresses are the same, even when virtual memory is not turned on, will fail to operate properly.

Hardware Addresses

Hardware components, such as the PCI bridge and the interrupt controller, are not located at the same addresses as on previous PCI-based Macintosh computers. The Name Registry provides the addresses. For information about the features of the Macintosh Name Registry, see "Using the Name Registry" (page 285).

Macintosh Name Registry

The Macintosh Name Registry contains the information in the updated Open Firmware device tree. This functionality is unchanged by the NewWorld architecture except to provide new mechanisms that provide communication with Open Firmware. The Open Firmware config variables used by Open Firmware during boot are modifiable from the Mac OS by using the existing Name Registry API calls. Properties saved in NVRAM (nonvolatile RAM) by the Mac OS are available to Open Firmware FCode drivers as well as Mac OS runtime drivers. See "New Name Registry Functionality" (page 79) for more details.

The gestaltMachineType Value

All NewWorld-based Macintosh computers return the same value (406 decimal) for the `gestaltMachineType`. Any software that depends on `gestaltMachineType` to verify that the current Macintosh computer is a valid CPU on which to execute needs to be changed to check for the existence of the required hardware nodes in the device tree. The Macintosh Name Registry provides routines for checking the device tree. The Name Registry is defined in Chapter 10, "Name Registry."

Interrupt Handling

Although the API calls related to interrupt handling have not changed, the code that handles interrupts is very different in the NewWorld architecture. The new interrupt code allows for dynamic creation of the interrupt layout. In addition, the interrupt latency has been reduced to such an extent as to make it negligible. The description of the interrupt layout is now part of an Open Firmware interrupt tree that is interlaced within the Open Firmware device tree. The Trampoline code, which assists in transferring startup control from Open Firmware to the Mac OS, uses this interrupt tree to build the Mac OS native interrupt tree.

ROM-in-RAM

The NewWorld architecture puts the Mac OS ROM image in RAM and marks it read-only. Although the image is 4 MB in size, not all of those 4 MB are in use.

The portion that is not used is returned to Mac OS for use as part of system RAM. At the time this document was written, about 3 MB of the 4 MB Mac OS ROM image are in use, allowing about 1 MB to be returned to Mac OS as available RAM.

The fact that ROM is stitched into RAM is the reason that the logical and physical memory addresses are no longer mapped one-to-one.

Runtime Abstraction Services (RTAS)

Certain hardware devices, such as custom I/O controllers, differ from Macintosh to Macintosh, but provide similar functions. RTAS communicates with the underlying hardware to provide services such as accessing the real-time clock, nonvolatile RAM (NVRAM), restart, shutdown, and PCI configuration cycles. The services provided by RTAS are not available to clients other than the Mac OS.

NVRAM and PRAM

Instead of using hard-coded offsets to locations in NVRAM for Mac OS PRAM and other information, NewWorld breaks NVRAM into variable-sized partitions that are used by the Mac OS, Open Firmware, and any other client. The partitioning scheme is part of the CHRP specification. PRAM resides in the Mac OS partition, and API calls to modify PRAM refer to offsets within that partition. Properties saved in NVRAM are saved in an Open Firmware config variable. See “New Name Registry Functionality” (page 79) section for additional details.

USB

The USB Manager is in a NewWorld Mac OS ROM image, along with class drivers for USB hubs, keyboards, and pointing devices.

ADB

Macintosh computers no longer always include ADB hardware. For compatibility, the ADB Manager still functions, treating USB keyboards as a variant of an ADB keyboard. This added compatibility does not allow all ADB devices to work as if ADB hardware still exists, even if a USB-ADB conversion device is attached to a USB connector.

Floppy Drives

The NewWorld architecture supports Macintosh computers with or without floppy drives. Apple does not support floppy-based copy protection on Macintosh computers that do not have Apple floppy drives. Providing functionality similar to the Apple floppy driver is the responsibility of the developer of the software for mass storage devices that can read and/or write floppy diskettes. The most expedient way to provide such functionality is to take over the Apple floppy driver slot in the drive queue.

Video Drivers

To present a seamless transition between the Open Firmware user interface and the user interface used by the Mac OS, the NewWorld architecture provides a mechanism for communication of the display mode, resolution, and so on, between the Open Firmware video driver and the Mac OS video driver. See "New Name Registry Functionality" (page 79) for additional details.

NewWorld Components

The three main areas of change, which can be thought of as new components, in the NewWorld architecture are the bootROM, the Mac OS ROM bootinfo file, and the changes in Mac OS system software. Before the NewWorld architecture, the Macintosh ROM contained all of the hardware-specific initialization code, the Mac OS—specific start-up code, and the Toolbox functions. The NewWorld architecture breaks up that monolithic Macintosh ROM into several pieces that are located in two places:

- The bootROM is a physical part of the specific implementation of current Macintosh computers. It contains these pieces:
  - POST (Power-on Self Test), startup code without Mac OS—specific code
  - Open Firmware
  - RTAS (Runtime Abstraction Services)
  - Mac OS drivers (of type 'ndrv' and 'nlib') for logic board I/O controllers and devices needed at boot time
- The Mac OS ROM image file is kept in the System Folder of the startup volume. It contains these pieces:
  - Mac OS—specific Open Firmware code and required bootinfo components

- □ Open Firmware—specific Mac OS code that takes care of transferring control from Open Firmware to the Mac OS ROM and system boot services
- □ a Mac OS ROM image, including among other things, the Macintosh Toolbox code

Other operating systems may also use the bootROM and a bootinfo file, but the contents of the bootinfo file are specific to the operating system that boots after logic board and hardware initialization by the code in the bootROM. The contents of the Mac OS ROM bootinfo file listed above are specific to Mac OS implementations.

The bootinfo file exists on the boot device and has a localizable name. Identification information that leads to the file's path is stored in the Open Firmware config variables in NVRAM. The search algorithm for a usable bootinfo file parallels the search mechanism across SCSI, ATA, and other interfaces used in the older Mac OS start-up implementations. The path is specified in a string that looks similar to this:

```
/pci/mac-io@10/ide@20000/disk@0:5,\System%20Folder\:tbxi
```

The elements that make up the node portion of the path, everything up to `disk@0:`, are defined in Section 3.2.2.1 "Node Names" in the IEEE 1275 specification. The remaining elements in the path string are file specific.

By default, the bootinfo file is located by using the blessed folder directory ID `dirID` in the master directory block, and then searching for a file with a file type of `'tbxi'`. Searching by file type is done to allow localization of the filename. The name of the nonlocalized bootinfo file is currently Mac OS ROM. This name may change in the future.

In order for a mass storage drive to be bootable, Open Firmware needs methods for accessing the device at the block level. The Open Firmware `deblocker` and `disk-label` packages are useful for providing boot-time services. For standard built-in devices, such as SCSI and ATA, methods are supplied by Open Firmware. For plug-in expansion cards, such as PCI cards, the Open Firmware methods must be supplied in the expansion ROM for the card and must follow the rules defined by the PCI Open Firmware binding specification. See "Sample FCode Drivers" (page 131) for a basic example of an FCode block device driver.

Some versions of the Mac OS ROM bootinfo file also contain what has traditionally been part of an enabler. This is only to reduce the number of files in the System Folder, and Open Firmware does not use the enabler components in any way.

A bootinfo file contains Open Firmware script, a description, information for individual operating systems, icons, and other information. A bootinfo file can be extended to contain non—Open Firmware information, which in the case of the Mac OS, is the addition of the Macintosh Toolbox and other code specifically for use by the Mac OS.

Note
The PowerPC™ Microprocessor Common Hardware Reference Platform (CHRP) System binding to: IEEE Std 1275-1994: Standard for Boot (Initialization, Configuration) Firmware document provides additional information about how a bootinfo file is used. However, it is not the reference specification for the bootinfo file currently in use on the Macintosh platform. ◆

NewWorld Boot Process

The following list provides a high-level overview of the execution path taken when a NewWorld-based computer boots the Mac OS:

1. User presses power key. Between the time that the power key is pressed and the boot beep is heard, while the screen is still black, a ROM checksum is taken, the processor is checked, the interrupt controller is started, all the clocks are determined, the memory controller is initialized, NVRAM is checked, RAM is sized checked and initialized, and the L2 cache is sized and prepared (L2 cache is enabled in POST).
2. The POST code runs (preliminary diagnostics, boot beep, initialization, and setup). This is like similar code in an older Macintosh ROM, but it is different in that it does not contain code specific to an operating system.
3. Open Firmware initializes begins probing the hardware and the PCI bus to locate attached interfaces and hardware so that is can begin building the device tree.
4. Open Firmware loads the Mac OS ROM image file, based on defaults and the path settings found in NVRAM.
5. Open Firmware executes the Forth script in the bootinfo file, which contains information about the rest of the file and instructions to read both the Trampoline code and the Mac OS ROM file and place them into a temporary place in memory.

6. The Forth script transfers control to the Trampoline code, which functions as the transition between Open Firmware and the beginning of the Mac OS execution.
7. The Trampoline code gathers information about the system from Open Firmware, creates data structures based on this information, terminates Open Firmware, and moves the contents of memory to an interim location in physical memory space.
8. The Trampoline code transfers control to the Mac OS ROM initialization code.

The boot sequence, up to loading and execution of the Mac OS ROM bootinfo file, is controlled by Open Firmware. To provide a user experience like that of previous Macintosh computers, Open Firmware supports keyboard input for searching for possible boot devices, and user redirection of boot devices. The keyboard input is done by holding down a specific key or key combinations. These keys are referred to as snag keys. The supported snag keys are listed in Table 3-1.

Table 3-1 Snag keys supported by Macintosh Open Firmware

Key or key combination	Definition
c	Redirect booting to the device alias CD. If no bootable partition is found on the CD, display blinking folder.
d	Redirect booting to the device alias hard drive. If no bootable partition is found on the hard drive, display blinking folder.
n	Redirect booting to the device alias `enet` for BOOTP or TFTP network booting.
z	Redirect booting to the device alias ZIP. If no bootable partition is found on the Zip drive, display blinking folder.
Command-option-shift-delete	Redirect booting to any device other than the device specified in the device alias boot-device.

When the user selects a startup device in the Startup Disk control panel, the control panel no longer sets a value in Mac OS PRAM, but rather generates an Open Firmware path to the device and saves that path in NVRAM as Open Firmware's `boot-device` config variable. Open Firmware tries booting from the device specified by `boot-device` first. If this device is unavailable or the user has overridden the standard startup behavior with keyboard input, Open Firmware scans other devices looking for bootable drives. Once Open Firmware selects a device, it reports a path to that device in the `bootpath` property in the chosen device tree node. The `bootpath` property is what the Mac OS ROM subsequently uses to locate and load the Mac OS from disk.

Additional information about the requirements for participating in the boot process on a NewWorld-compatible Macintosh can be found in Chapter 4, "Startup and System Configuration."

New Name Registry Functionality

NewWorld-compatible Macintosh computers have a greater dependence on Open Firmware and require additional functionality in the Mac OS. The Name Registry continues to provide a database that includes the device tree used by Open Firmware, Open Firmware drivers, Mac OS device services, and PowerPC native device drivers. NewWorld extends the Name Registry API to provide

- a mechanism to update Open Firmware config variables
- communication between Mac OS and Open Firmware drivers

Modifying Open Firmware Config Variables From the Mac OS

The Name Registry provides a mechanism to save and restore device tree properties in NVRAM. This mechanism has been fully augmented in the NewWorld architecture. Using the same mechanism, Mac OS and Open Firmware driver software can update Open Firmware config variables in the device tree. The mechanism creates and modifies the device tree properties and the `RegistryPropertySetMod` function, using the `kRegPropertyValueIsSavedToNVRAM` modifier bit. When the Name Registry calls are made to the properties in the "device-tree:options" node, the corresponding Open Firmware config variables in NVRAM are modified.

For additional information about Name Registry properties, see "Property Management" (page 311). For information about the `kRegPropertyValueIsSavedToNVRAM` modifier bit, see "Data Structures and

Constants" (page 326). For a description of the `RegistryPropertySetMod` function, see "Property Modifier Retrieval and Assignment" (page 332).

Communication Between Mac OS and Open Firmware Drivers

In addition to providing access to Open Firmware config variables, the same Name Registry mechanism that saves and restores device tree properties in NVRAM has new functionality that allows communication between a Mac OS runtime driver and its corresponding Open Firmware driver. The Mac OS runtime driver sets a property in its device tree node by setting the `kRegPropertyValueIsSavedToNVRAM` modifier bit. This property is saved in a special Open Firmware config variable that is used by Open Firmware during boot to restore properties into the device tree. Since the properties exist when an Open Firmware driver is opened, such a property can be a message from the Mac OS runtime driver to the Open Firmware driver. The Mac OS driver will also find the property during its initialization. The new mechanism extends the size limitations for properties saved in NVRAM to 8 bytes for the name and 32 bytes for the data.

CHAPTER 4

Startup and System Configuration

This chapter describes the Open Firmware startup process by which Power Macintosh computers recognize and configure peripheral devices connected to the PCI expansion card bus. The Open Firmware process provides flexibility in system software to match the flexibility that the PCI bus provides for expansion hardware.

The PCI bus architecture described in the PCI standard supports the **autoconfiguration** concept of system configuration, because it includes mechanisms for configuring devices during system startup and defines expansion ROMs for plug-in expansion cards. The two code types currently defined for PCI expansion card ROMs are an Intel-compatible BIOS (Basic Input and Output System) code type and the Open Firmware type. Apple has chosen the Open Firmware type because it allows Power Macintosh computers to run nearly any operating system.

The iMac and computers introduced after the iMac implement the NewWorld system architecture. The NewWorld architecture relies explicitly on the Open Firmware startup boot process for devices attached to the PCI bus and other I/O buses. Peripheral devices that need to participate in the boot process at startup time in NewWorld computers must support Open Firmware as described in this chapter. An introduction to the details of the NewWorld architecture is provided Chapter 3, “Introduction to the NewWorld Architecture.”

Peripheral Devices and Open Firmware

The PCI bus gives Power Macintosh computers a increased compatibility with third-party hardware devices. To provide equivalent software compatibility for all device I/O, Power Macintosh computers that implement the PCI bus standard, also support the IEEE standard Open Firmware process of system startup.

During the Open Firmware startup process, startup code in the Macintosh computer’s bootROM searches the PCI and other I/O buses, such as USB or FireWire, and generates a data structure called a device tree that lists all available peripheral devices. This data structure also stores information about the support software, including drivers, provided by each PCI expansion card. The startup code then finds information that points to an operating system either in ROM or on a mass storage device, loads it, and starts it running. The operating system does not need to be Mac OS. Hence it is possible for Power

Macintosh computers to operate PCI peripheral devices using either Macintosh or third-party system software.

A PCI card that wants to participate in the startup process of any operating system must include an expansion ROM containing an Open Firmware FCode driver, methods, and properties. Examples of expansion cards that need to operate in the Open Firmware startup process are SCSI cards, Ethernet cards, and display cards. The alternatives for FCode in expansion card ROMs are described in "Open Firmware FCode Options" (page 87).

The Open Firmware Startup Process

The **Open Firmware startup process** in Power Macintosh computers with a PCI bus conforms to IEEE Standard 1275 and to the *PCI Bus Binding to IEEE 1275-1994* specification. These standards evolved from the OpenBoot firmware architecture introduced by Sun Microsystems. The *PCI Bus Binding to IEEE 1275-1994* specification is available at http://bootrom.apple.com/1275/home.html#OFDbusPCI. The *IEEE Std 1275-1994: IEEE Standard for Boot Firmware (Initialization Configuration) Firmware: Core Requirements and Practices* is available at http://standards.ieee.org/index.html.

Note
The 1275 Working Group continues to update the *PCI Bus Binding to IEEE 1275-1994* specification. For latest information, you can access the FTP site listed in the note under "Institute of Electrical and Electronic Engineers" (page 27). ◆

Additional information about the Open Firmware startup process for PowerPC computers can be found in the *PowerPC™ Microprocessor Common Hardware Reference Platform (CHRP™) System binding to: IEEE Std 1275-1994 Standard for Boot (Initialization, Configuration) Firmware, Revision: 1.8 [Approved Version]* at http://bootrom.apple.com/1275/

Startup Firmware

The Open Firmware startup process is driven by **startup firmware** (also called *boot firmware*) in the **Power Macintosh ROM** and in memory chips on expansion cards, called **expansion ROMs.**

Note
Power Macintosh computers that implement the NewWorld architecture (iMac and later) have the startup code in a small **bootROM**. The remainder of what has been traditionally referred to as the Macintosh Toolbox is loaded from a file, called Mac OS ROM, located on disk, and then stitched into RAM to appear as one contiguous read-only ROM. Power Macintosh computers built before the NewWorld implementation have the startup code and the Macintosh Toolbox code in a larger ROM, usually on the order of 4 MB in size. ◆

While the Open Firmware startup code is running, the Power Macintosh computer starts up and configures its on-board hardware, including all peripheral devices the startup code knows about or has config variables for, independently of any operating system. The computer then finds an operating system in ROM or on a mass storage device, loads it into RAM, and terminates the Open Firmware startup process by giving the operating system control of the PowerPC processor. The operating system may be Mac OS or a different system, provided it uses the PowerPC instruction set.

The Open Firmware startup process includes these specific features:

- Startup firmware, as defined by IEEE Standard 1275, is written in the Forth language. Firmware code is stored in an abbreviated representation called **FCode,** a version of Forth in which most Forth words are replaced by single bytes or 2-byte groups. The startup firmware in the Power Macintosh ROM provides an FCode loader that installs FCode in system RAM that the Open Firmware Forth interpreter executes. Expansion card firmware can modify the Open Firmware startup process by supplying a FCode boot driver that the computer's startup firmware loads and runs before launching an operating system. The expansion card startup firmware can also include a minimal run-time driver that works in the Open Firmware environment and is replaced by a comprehensive driver later in the boot process.
- The Macintosh startup firmware creates a hierarchical data structure of nodes called a **device tree,** in which each device is described by a **property list**, and optional properties and methods. The device tree is stored in system RAM. The nodes, which are also called packages, contain properties and methods. Properties are attributes that describe the hardware and driver. Methods do work much like subroutines or procedures. The operating system that is ultimately installed and launched can search the device tree to determine what hardware is available. For example, Mac OS extracts

information from the device tree to create the device portion of the Macintosh Name Registry, described in Chapter 10. The full list of standard device tree properties is given in IEEE Standard 1275; the properties that Mac OS uses are listed in Table 10-1 (page 322). An example of a device node in a device tree is given in Listing 10-1 (page 284).

- Device drivers that are required during system startup are also written in FCode. Plug-in expansion cards for startup devices must contain all the driver code required during startup in the expansion ROM on the card and may also need to provide other driver support resources such as fonts. The startup firmware in the Power Macintosh ROM installs Open Firmware boot drivers in system RAM and the Open Firmware Forth interpreter executes the driver FCode. Examples of devices needed during system startup include display, keyboard, and mouse devices; storage devices such as SCSI, IDE, floppy disk, and magneto-optical drives; and network interfaces if the target OS supports network booting.

You can write PCI expansion ROM code in standard Forth words and then reduce the result to FCode by using an **FCode tokenizer,** a program that translates Forth words into FCodes. The Forth vocabulary that you can use is presented in IEEE Standard 1275. For a description of the Open Firmware user interface, see "Open Firmware User Interface" (page 114).

Device Drivers

The Open Firmware startup process and all possible operating systems constitute separate **device environments.** A separate driver is normally required for each device environment in which a device is expected to work. In rare cases, an operating system may be written so that it uses an Open Firmware driver or a driver for another operating system.

The following rules govern the requirements for device drivers in Power Macintosh computers that support the Open Firmware startup process:

- As explained in the previous section, Open Firmware drivers must be stored as FCode in a card's expansion ROM and must conform to IEEE Standard 1275.
- A card's expansion ROM should also contain all the run-time drivers for different operating systems that might use or support the card.
- If an operating system preserves and uses the Open Firmware device tree or a data structure derived from it, it should store all device drivers specific to

that environment in the device tree as properties of the devices they support. Otherwise the operating system must load device drivers as part of its initialization.

- Drivers that work with Mac OS must be compiled to native PowerPC code. For further information, see Chapter 8, “Writing Native Drivers.”

Chapter 5, “PCI Open Firmware Drivers,” provides guidelines for writing device drivers to operate with the Open Firmware startup process.

PowerPC Addressing and Alignment

In general, PCI expansion cards that run code directly on PowerPC processors in Power Macintosh computers must use 32-bit mode even when the processor supports 64-bit mode. PCI cards must observe the access sizes and byte alignments shown in Table 4-1.

Table 4-1 PowerPC processor addressing

Open Firmware Address type	Access size (bits)	Alignment (bytes)
`a-addr`	32	4
`q-addr`	32	4
`w-addr`	16	2

Device Configuration

PCI cards should supply Open Firmware boot code in PCI type 1 containers in their expansion ROMs, as defined in the PCI specification. This section describes how the contents of PCI expansion ROMs contribute to the Open Firmware startup process.

Open Firmware FCode Options

Cards that may be required during Open Firmware startup include display, keyboard, and mouse devices, mass storage devices such as SCSI, IDE, floppy disk, and magneto-optical drives, and network interfaces. If Open Firmware FCode is not included in such a card's expansion ROM, the card will not be usable until the operating system loads its supporting software from a mass storage device after startup.

This section describes the possible ways that a device with a valid PCI expansion ROM can be configured. They range from full Open Firmware support, in which the card is usable during startup, to no support.

Full Open Firmware Support

The recommended option is for every PCI card to include an expansion ROM containing run-time drivers and full Open Firmware support, including Open Firmware properties and FCode boot driver software that supports the startup process. With this option, the associated device can be used at startup time by Open Firmware and by any operating system for which the PCI card's expansion ROM provides a native run-time driver.

IMPORTANT

Full Open Firmware support in the expansion ROM is mandatory for PCI cards supporting devices that expect to participate in the startup boot process on Macintosh computers designed around the NewWorld architecture. ▲

An expansion ROM that supports Open Firmware booting delivers these benefits:

- full plug-and-play performance with any operating system for which the card provides a run-time driver
- unambiguous matching of each run-time driver to its device

Support for Mac OS and Open Firmware

A less desirable option is for the PCI card to include an expansion ROM containing a Mac OS run-time driver and minimum Open Firmware support, including Open Firmware properties. This option lets the card work during startup with Mac OS running on PCI-based Power Macintosh computers introduced prior to the iMac, where startup is controlled by the Macintosh

ROM. The card will not work during startup on current Power Macintosh models. This option delivers these benefits:

- full plug-and-play performance with the Mac OS
- unambiguous matching of the Mac OS run-time driver to the device

Minimum Open Firmware Support

The minimum recommended option is for the PCI card to include an expansion ROM that provides minimal Open Firmware support, including the Open Firmware properties `name`, `reg`, and `device type`. This option provides enough information to allow Open Firmware to build a name property for the device that is guaranteed to be unique, so the Mac OS can match it unambiguously to a run-time driver that it loads from the Extensions folder. This option does not include a FCode driver to allow your device to provide any services at boot time before the operating system is running.

No Open Firmware Support

The least desirable option is for the PCI card to include an expansion ROM with no FCode, or no expansion ROM at all. At system startup time, the card is recognized and address space is allocated for the device, but no peripheral initialization or driver code is loaded. The operating system must load driver code from a mass storage device before the card can be used. Most importantly, there is no distinct name property for the device; this makes unambiguous run-time driver matching less certain when several card manufacturers support the same device. Driver matching issues are discussed in "Matching Drivers With Devices" (page 164).

Note
Because future Macintosh computers may run a variety of operating systems, full Open Firmware support is particularly important for PCI-based graphics devices. If a PCI device is the user's only display, it should operate during the Open Firmware startup process and should deliver plug-and-play performance with the user's choice of operating system. The Open Firmware driver does not need to be sophisticated; if it can initialize the device to 8-bit color mode and publish the frame buffer address, Open Firmware in the bootROM will control the device and perform the required image rendering. ◆

Open Firmware Driver Support

As explained in "Startup Firmware" (page 83), Open Firmware drivers are stored as FCode in expansion ROMs and copied into system RAM during the Open Firmware startup process. When the startup firmware in the Power Macintosh ROM opens an Open Firmware driver, it acquires a handle to the driver code so it can communicate directly with it. The Power Macintosh firmware provides three kinds of memory for the driver to use:

- The device tree stores properties and routines that are intrinsic to the driver; these permanent attributes are always available to the driver and other code.
- Each node of the device tree has its own static variables, available to drivers, which are preserved throughout the Open Firmware startup process.
- Memory for buffers and other driver requirements is allocated each time a driver is opened and is maintained until the driver is closed.

Open Firmware drivers are expected to perform their work (such as drawing characters on a screen) without operating-system support. However, the startup firmware in some Power Macintosh ROMs may contain hardware-specific support packages that Open Firmware drivers can use for common tasks. The supported packages are located in the node `/packages`. The Macintosh Open Firmware does not provide interrupt service routines (ISR) for handling hardware interrupts; Open Firmware drivers must detect external events by polling devices.

Startup Sequence

This section defines the startup sequence for Power Macintosh computers built prior to the iMac. The iMac and all Macintosh computers built after the introduction of the iMac are designed to support the NewWorld architecture. The startup sequence for NewWorld based computer models differs from that of previous Macintosh computers. And, it does so at no loss of software compatibility with well behaved applications. See "Startup Sequence in the NewWorld Architecture" (page 92) for a high-level description of the NewWorld startup sequence.

Although the startup sequence for PCI-based Power Macintosh computers is different for each model, a typical sequence for a Power Macintosh computer running Mac OS can be summarized as follows, starting with the power coming on:

1. System-specific firmware performs initialization and self-testing on memory and other hardware systems.
2. The startup firmware in the Power Macintosh ROM probes each PCI bus, generates a device tree node for each device, and when it finds a device executes the FCode (if any) found in each PCI card's expansion ROM.
3. The startup firmware in the Power Macintosh ROM finds an operating system in ROM or on a mass storage device; it loads it into RAM and transfers processor control to it.
4. Mac OS completes the startup sequence.

The rest of this section describes these steps in more detail.

Initializing the Hardware

In response to power coming on, firmware in the Power Macintosh ROM performs initialization and self-testing on the basic system memory, including RAM and cache memory.

Running Open Firmware

The Open Firmware Process begins as the startup firmware builds the device tree for built-in I/O devices and then searches expansion areas for other devices. The firmware polls the computer's PCI buses, interrogating addresses where devices might be found. Each time it finds a device with an Open Firmware expansion ROM, it copies the FCode from that ROM into system RAM and executes it, using the system's FCode loader. As it runs, the FCode program enters the properties of the device it represents into the current device tree node established by the Open Firmware program and stored in system RAM.

An important set of device tree properties include Open Firmware drivers for PCI devices. Run-time drivers, which are stored as properties of the device node in the device tree, may be required for the startup process and for each operating system that may be launched. Other properties include operating characteristics of video cards and information used to install interrupt handlers.

Open Firmware queries PCI cards that contain no FCode and creates basic node entries for them in the device tree. These nodes contain only the properties that can be generated by accessing a card's standard PCI configuration registers. Open Firmware creates `reg` and `assigned-addresses` properties, making the card accessible to operating-system code (although not to Open Firmware). These properties provide the card's unit address and permit address space allocation based on the card's PCI base register support. PCI properties are discussed in "Standard Properties" (page 322).

Starting the Operating System

After constructing the device tree in system RAM, the Power Macintosh startup firmware selects some or all of the following startup devices, based on an order of priority stored in the system hardware and on the presence of suitable device properties in the device tree:

- a keyboard (or other input device)
- a display (or other output device)
- a boot device (mass storage or ROM, indicated by the boot path environment variable) that contains operating-system code

The Open Firmware code normally loads the operating system into memory and starts it using the Forth `go` command. In the case of Mac OS, Open Firmware transfers processor control to the Mac OS ROM, which begins the Mac OS startup process. If the Open Firmware user interface is invoked, however, the Open Firmware code will continuously poll the input device for characters and write output characters to the display, using the FCode drivers previously installed. This can let the user choose an operating system or perform other OS-independent configuration tasks. For further details, see "Open Firmware User Interface" (page 114).

For further details of the normal Macintosh startup sequence, see Chapter 10 of *Technical Introduction to the Macintosh Family,* described in "Supplementary Documents," in the preface.

Startup Sequence in the NewWorld Architecture

Here is a high-level view of the execution path taken when a NewWorld-based computer starts up.

1. The POST code runs (preliminary diagnostics, boot beep, initialization, and setup), with possible intervention in the mini nub, a small debugging tool.
2. Open Firmware initializes and begins execution, including building the device tree and the interrupt trees.
3. Open Firmware loads the Mac OS ROM file, based on defaults and NVRAM settings.
4. Open Firmware executes the Forth script in the Mac OS ROM file, which contains instructions to read both the Trampoline code and the compressed Mac OS ROM image and place them into a temporary place in memory.
5. The Forth script transfers control to the Trampoline code, which functions as the transition between Open Firmware and the beginning of the Mac OS execution.
6. The Trampoline code decompresses the Mac OS ROM image, gathers information about the system from Open Firmware, creates data structures based on this information, terminates Open Firmware, and rearranges the contents of memory to an interim location in physical memory space.
7. The Trampoline code transfers control to the `HardwareInit` routine in the Mac OS ROM bootinfo file.
8. The `HardwareInit` routine copies data structures to their correct places in memory, and then calls the NanoKernel.
9. The NanoKernel fills in its data structures and then calls the 68K emulator.
10. The 68K emulator initializes itself, then transfers control to the startup initialization code.
11. The startup initialization code begins execution, initializing data structures and managers, and booting the Mac OS.

All functions found in the old Mac OS ROM are present in the NewWorld boot process, but occur at different times and places. To accomplish this, the code in the Mac OS ROM Image and POST is simplified, while the Trampoline code addresses the new functionality.

What Is Different

Even though ROM-in-RAM involves a fundamental change to the construction of the product-specific part of the Mac OS, the changes in the code and its execution are not that large. Many components are in changed locations, but their functions with respect to boot time and run time have not greatly changed. Many Mac OS components remain untouched.

Interrupt Handling

Interrupt handling is very different with the NewWorld approach. The interrupt code has been rewritten to allow for dynamic creation of the interrupt layout. In addition, interrupt latency has been reduced to such an extent as to make it negligible. The description of the interrupt layout is now part of an Open Firmware interrupt tree that is interlaced within the Open Firmware device tree. The Trampoline code uses this interrupt tree to build the Mac OS native interrupt tree.

RAM Footprint

The NewWorld architecture puts the Mac OS ROM Image in RAM, and marks it read-only. Although the image is 4 megabytes in size, not all of it is in use. The portion that is not used is returned to the Mac OS for use as part of system RAM. At the time this document was written, less than 3 megabytes of the 4 megabyte Mac OS ROM Image are in use, allowing more than 1 megabyte to be returned to the Mac OS.

Run-Time Abstraction Services (RTAS)

Certain hardware devices differ from machine to machine, but provide similar functions. RTAS provides such hardware-specific functions, including functions for accessing the real-time clock, nonvolatile RAM (NVRAM), restart, shutdown, and PCI configuration cycles. RTAS is not available to clients other than the Mac OS.

NVRAM

Instead of using hard-coded offsets to locations in NVRAM for Mac OS NVRAM and other information, the Trampoline code breaks NVRAM into variable-sized partitions that are used by Mac OS, Open Firmware, and any other client. PRAM resides in the Mac OS partition. The partitioning scheme is defined in part by the Common Hardware Reference Platform (CHRP) specification.

Startup Disk Control Panel

Open Firmware now bears responsibility for locating a startup device. This is very different from previous Mac OS systems where the Mac OS ROM had responsibility for locating the startup device. On the Power Macintosh G3 computer, the Mac OS ROM image itself comes from the startup disk, so decisions regarding startup device must be made earlier in the startup process. Open Firmware recreates as much as possible the user experience of earlier systems but the implementation is very different.

Previous systems stored the user's selected startup device in PRAM. The startup device was set in NVRAM when the user selected a device in the Startup Disk control panel. This device was honored by the Mac OS ROM unless the selected device was unavailable or was overridden by the user.

The startup disk routine for the NewWorld computers sets an Open Firmware configuration variable called `boot-device`, rather than setting Mac OS PRAM. This setting is honored by Open Firmware unless the selected device was unavailable or was overridden by the user.

The following keys can be used to override the selected startup device.

- Command-Option-Shift-Delete: ignore the boot-device setting and scan for alternate devices.
- C: force the internal CD-ROM drive to be the startup device
- D: force the internal hard disk to be the startup device
- N: force boot using BOOTP and TFTP over the Ethernet
- Z: force the internal Zip drive to be the startup device

Once Open Firmware locates a startup device and successfully loads a Mac OS ROM image, it passes information about the chosen device in the bootpath variable. This information, rather than that previously set in PRAM, is subsequently used by the Mac OS ROM to locate the device containing the startup System Folder.

IMPORTANT
The previous API for controlling the startup device selection, using `_GetDefaultStartup` and `_SetDefaultStartup`, is not effective on computers that support the NewWorld architecture. ▲

PCI Bus Configuration

This section describes how the Power Macintosh Open Firmware code configures the computer's PCI buses during the Open Firmware startup process.

Configuration Tasks

Macintosh Open Firmware code performs the following tasks to help the PCI system support the devices previously found by the Open Firmware startup process:

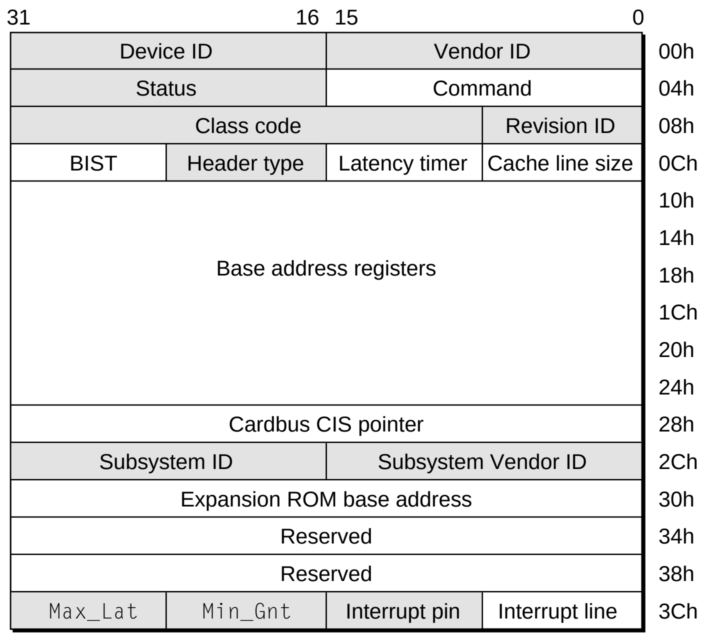
- It programs certain configuration bits in the 64-byte standard PCI header portion of PCI configuration space.
- It determines the resource requirements (memory and I/O space) of each device, based on the device's `reg` property. The `reg` property is created either by executing the FCode in the card's expansion ROM, or if FCode is not present, the system Open Firmware code creates a `reg` property for the device by querying the device's PCI configuration base registers.
- After accumulating the resource requirements for all devices in the system, the system Open Firmware code constructs a conflict-free address map and adds the resulting `assigned-addresses` property to each PCI device's node in the device tree.

Configuration Registers

Figure 4-1 presents a map of the PCI configuration registers that system firmware reads or writes to during the Open Firmware startup process. In

Figure 4-1, read-only registers are shaded; all other registers are read/write. The next section describes the actions taken for each register.

Figure 4-1 PCI configuration register map



Register Actions

This section describes the actions that the Macintosh Open Firmware performs on the PCI configuration registers listed in Figure 4-1 during startup.

Vendor ID

The Vendor ID register is read and its value stored in the property `vendor-id`. If the card has no FCode, the Vendor ID value makes up the *xxxx* portion of the "`pci`*xxxx,yyyy*" default name property for the node.

Device ID

The Device ID register is read and its value stored in the property `device-id`. If the card has no FCode and no subsystem ID, the Device ID value makes up the *yyyy* portion of the "`pci`*xxxx*,*yyyy*" default name property for the node.

Command

The following bits in the Command register are set with the meanings shown:

- Bit 9, Fast Back-to-Back Enable, is set to 1 if all PCI devices are fast back-to-back capable (if all devices have a fast-back-to-back property stored in their device node); otherwise, it is cleared to 0.
- Bit 8, SERR Enable, is cleared to 0 for all devices because the Power Macintosh system doesn't respond to SERRs.
- Bit 7, Wait Cycle Control, is cleared to 0 for all devices.
- Bit 6, Parity Error Response, is cleared to 0 for all devices.
- Bit 5, VGA Palette Snoop, is cleared to 0 for all devices.
- Bit 4, Memory Write and Invalidate Enable, is set to 1 for all devices because the Power Macintosh system fully supports this command type and optimizes for it.
- Bit 3, Special Cycle Enable, is set to 1 for all devices because the Power Macintosh system can generate special cycles.
- Bit 2, Bus Master Enable, is set to 1 for all devices because the Power Macintosh system supports masters in all PCI locations.
- Bit 1, Memory Space Enable, is cleared to 0 for all devices before an operating system is loaded. Hence, the initialization routines of all run-time drivers must set this bit to 1 if they wish to access their device in memory space. However, the decision to write a 1 in this location must be made after checking that the memory resources required for correct operation appear in the device's `assigned-addresses` property; otherwise, the driver should leave this bit to cleared to 0.
- Bit 0, I/O Space Enable, is cleared to 0 for all devices before an operating system is loaded. Hence, the initialization routines of all run-time drivers must set this bit to 1 if they wish to access their device in I/O space. However, the decision to write a 1 in this location must be made after checking that the I/O space resources required for correct operation appear

in the device's `assigned-addresses` property; otherwise, the driver should leave this bit to cleared to 0.

Status

The following bits are read in the Status register:

The value of bits 10–9, DEVSEL Speed, is stored in the node's `devsel-speed` property.

The value of bit 7, Fast Back-to-Back Capable, is noted for each PCI device. If the value is nonzero, the property `fast-back-to-back` is created for the node. See the previous section for information about the Fast Back-to-Back Enable bit.

No specific action is taken for the remaining bits in the Status register.

Revision ID

The Revision ID register is read and its value stored in the property `revision-id`.

Class Code

The Class Code register is read and its value stored in the property `class-code`.

Cache Line Size

The Cache Line Size register is set for all devices as specified in the PCI Specification 2.1. This value may change from Macintosh platform to Macintosh platform for various performance reasons.

Latency Timer

The Latency Timer register is set for all devices as specified in the PCI Specification 2.1. This value may change from Macintosh platform to Macintosh platform for various performance reasons.

Header Type

The Header Type register is read, starting with bits 6–0. If the value of bits 6–0 is 0x00, the configuration space has a standard header layout for configuration addresses 0x10 through 0x3F; if the value is 0x01, it has a PCI-to-PCI bridge header layout for that section.

Note
The PCI bus behavior described in this section is that corresponding to a standard header. ◆

If bit 7 of the Header Type register is set to 1, the system Open Firmware probes for multiple functions; otherwise, it assumes the device is a single-function device.

BIST

No action is taken on the BIST register.

Base Registers

If FCode is present in the card's expansion ROM, the system Open Firmware creates an `assigned-addresses` property for the node, provided the card's FCode presents a `reg` property with entries referencing at least one base register and the system was able to provide the resources requested in the `reg` property corresponding to the base registers referenced. For each base register that has a corresponding entry in the `assigned-addresses` property, the system Open Firmware programs that base register with the address value stored in the `assigned-addresses` property.

If FCode is not present for the node, the system Open Firmware creates a `reg` property for the device. To create a `reg` entry for each base register that is implemented, the system Open Firmware writes all 1s to each base register location. It then reads the locations to see how many of the 1s are still there. If the register reads back as all 0s, then the register is not implemented and a `reg` entry is not made for it. If the register contains a value other than 0, the system Open Firmware notes which bits are 1s and thereby determines whether the register is of type memory or I/O, the amount of address space required, whether it is a 64-bit address, whether it is prefetchable, and whether it must be located below 1 MB. This information is then encoded appropriately into the `reg` entry for the base register. After all base registers are queried in this manner, the full `reg` property is stored in the device's node. Refer to the PCI specification and *PCI Bus Binding to IEEE 1275-1994* (described in "Other Publications" (page 26)) for more details. Once the `reg` property is stored in the node, Open Firmware clears the Base registers to all 0s. It then follows the process of writing the registers with `assigned-addresses` values, as described above for devices that have FCode.

Subsystem Vendor ID

If the value of the Subsystem Vendor ID register is nonzero, a `subsystem-vendor-id` property is created with the register's value. If the property is created and no FCode is present on the card, the Subsystem Vendor ID value makes up the *xxxx* portion of the `"pci`*xxxx,yyyy*`"` default name property for the node.

The Subsystem Vendor ID register is described in Revision 2.1 of the PCI Specification.

Subsystem ID

If the value of the Subsystem ID register is nonzero and a `subsystem-vendor-id` property exists for the device, a `subsystem-id` property is created with the register's value. If the property is created and no FCode is present on the card, the Subsystem Vendor ID value makes up the *yyyy* portion of the `"pci`*xxxx,yyyy*`"` default name property for the node.

The Subsystem ID register is described in Revision 2.1 of the PCI Specification.

Expansion ROM Base

The system Open Firmware uses the Expansion ROM Base register at probe time to determine whether a card has FCode present. It queries the register to see whether the register is implemented, following the procedure described above for other base registers. If the register is implemented, Open Firmware temporarily maps in an amount of memory space equal to the requirement found from the base register query and then programs that value into the base register. It also enables the expansion ROM by an OR operation with 1 on bit 0 of the register and enables the card's memory space by writing a 1 to the correct bit in the Command register. It then reads the expansion ROM's first locations, by accessing the space temporarily mapped in, looking for the PCI signature (0x55AA). If it finds the signature, it continues to look for an Open Firmware ROM image signature. If it finds that signature, it locates the FCode, copies it to RAM, and executes it. After the card's FCode has finished executing, or if it was determined that there was no FCode, the system Open Firmware disables the card's memory space and expansion ROM and clears the Expansion ROM Base register to 0s.

If FCode was present in the card's expansion ROM and the FCode presented a `reg` property with an entry for the Expansion ROM Base register, and if the system was able to provide the resources for this entry, then the system Open

Firmware creates a corresponding entry in the `assigned-addresses` property and writes the address value to the Expansion ROM Base register.

If FCode is not present for the node, the system Open Firmware creates a `reg` property for the device and determines whether to create an entry for the Expansion ROM Base register following the procedure for other base registers described above. The procedure for writing the register if FCode is present is the same as that in the preceding paragraph.

IMPORTANT

Bit 0 of the Expansion ROM Base register, which is defined as the Expansion ROM Enable bit, is left as 0 (disabled) by the system Open Firmware. If the run-time driver is interested in accessing the PCI Expansion ROM, it must first check that it has received an `assigned-addresses` entry, and then it must enable both its memory space (Memory Space Enable bit of the Command register) and its ROM (Expansion ROM Enable bit of the Expansion ROM Base register). As with all writable configuration registers, such operations must be performed with read-modify-write code sequences so as not to disturb the existing values of other bits in the registers. ▲

Interrupt Line

No action is taken on the Interrupt Line register. It has no meaning for Power Macintosh computers because interrupts are OR-combined per slot in hardware, creating a unique interrupt for each PCI card accessible to the system interrupt controller. This register contains no useful information for drivers.

Interrupt Pin

The Interrupt Pin register is read. If its value is nonzero, the value appears in the property `interrupts`. This register contains no useful information for drivers for the reasons explained in the previous section.

Min_Gnt

The Min_Gnt register is read and its value stored in the property `min-grant`.

Max_Lat

The Max_Lat register is read and its value stored in the property `max-latency`.

PCI-To-PCI Bridges

The second generation of Power Macintosh computers implements PCI-to-PCI bridges in conformance with the PCI specification listed in "PCI Special Interest Group" (page 28).

Configuration Header

For PCI-to-PCI bridges, the standard PCI configuration header (the first 64 bytes of PCI configuration space) is different from that of standard PCI devices. Figure 4-2 gives a map of the registers in the portion of a PCI-to-PCI bridge's configuration space defined by the PCI specification. In Figure 4-2, read-only registers are shaded; all other registers are read/write.

Figure 4-2 PCI-to-PCI bridge register map

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class code			Revision ID	08h
BIST	Header type	Latency timer	Cache line size	0Ch
Base address registers				10h 14h
Secondary latency timer	Subordinate bus number	Secondary bus number	Primary bus number	18h
Secondary status		I/O limit	I/O base	1Ch
Memory limit		Memory base		20h
Prefetchable memory limit		Prefetchable memory base		24h
Prefetchable base upper 32 bits				28h
Prefetchable limit upper 32 bits				2Ch
I/O limit upper 16 bits		I/O base upper 16 bits		30h
Reserved				34h
Expansion ROM base address				38h
Bridge control		Interrupt pin	Interrupt line	3Ch

Register Settings

PCI-to-PCI bridges have specific configuration needs that are different from those of standard PCI devices. The system Open Firmware code is responsible for configuring the PCI-to-PCI bridge components. The following field descriptions list the standard settings for the registers shown in Figure 4-2.

Field descriptions

Vendor ID	Read by Open Firmware and stored in the property `vendor-id`.
Device ID	Read by system Open Firmware and stored in property `device-id`.
Command	Written by system Open Firmware. Bit specifics: Bit 9, Fast Back to Back Enable, is written 1 if all PCI devices are Fast Back to Back capable (if all devices have a

`fast-back-to-back` property stored in their device nodes); otherwise written 0.

Bit 8, SERR Enable, is written 0 for all devices; the Power Macintosh system doesn't respond to SERRs.

Bit 7, Wait Cycle Control, is written 0 for all devices.

Bit 6, Parity Error Response, is written 0 for all devices.

Bit 5, VGA Palette Snoop, is written 0 for all devices.

Bit 4, Memory Write and Invalidate Enable. PCI-to-PCI Bridges consider this a read-only bit and will always return 0 when read. They act only as agents for masters behind them and will propagate Memory Write and Invalidate commands if a PCI Master on either side generates such a cycle.

Bit 3, Special Cycle Enable. PCI-to-PCI Bridges consider this a read-only bit and will always return 0 when read, because they cannot respond to Special Cycles.

Bit 2, Bus Master Enable, is written 1 for all devices; the Power Macintosh system supports masters in all PCI locations.

Bit 1, Memory Space Enable, is written 1 for PCI-to-PCI bridges to enable memory cycles to pass through the bridge transparently, based on the programming of the Memory Base and Limit registers.

Bit 0, I/O Space Enable, is written 1 for PCI-to-PCI bridges to enable I/O cycles to pass through the bridge transparently based on the programming of the I/O Base and Limit registers.

Status — The following bits are read in the Status register:

Bits 10-9, DEVSEL speed, value stored in the node's `devsel-speed` property.

Bit 7, Fast Back to Back Capable, value noted for each PCI device. If the value is nonzero, the property `fast-back-to-back` is created for the node (see Command register explanation of Fast Back to Back Enable bit).

No specific action taken based on values of the remaining bits in the Status Register.

Revision ID	Read by system Open Firmware and stored in property `revision-id`.
Class Code	Read by system Open Firmware and stored in property `class-code`. The `name` property for PCI-to-PCI bridges defaults to `pci-bridg`e, based on the class code matching PCI-to-PCI bridge encoding (0x060400).
Cache Line Size	Written by system Open Firmware. Set to 0x08 for all devices, which corresponds to the PowerPC family cache line size of 32 bytes.
Latency Timer	Written by system Open Firmware. Set to 0x20 for all devices, which corresponds to 32 PCI clock intervals.
Header Type	Read by system Open Firmware. First, bits 6 through 0 are examined. If the value is 0x00, the configuration space has a standard header layout for configuration addresses 0x10–0x3F; if the value is 0x01, it has a PCI-to-PCI bridge header layout for that section. Described in this section is the behavior taken for a PCI-to-PCI header.
BIST	No action is taken by the system Open Firmware on this register.
base registers 0-1	Open Firmware does not set the Base Registers for PCI-to-PCI bridges. It is assumed that they are programmed only through PCI configuration space.
Primary Bus Number	Written by system Open Firmware with the appropriate PCI Bus number corresponding to this bridge's primary bus location (closer to main memory side) in the system topology.
Secondary Bus Number	Written by system Open Firmware with the appropriate PCI Bus number corresponding to this bridge's secondary bus location (farther from main memory side) in the system topology. This value is stored in the device tree as the first datum in the PCI-to-PCI Bridge's bus-range property.
Subordinate Bus Number	Written by system Open Firmware with the appropriate PCI Bus number corresponding to the highest numbered PCI bus that is located behind (subordinate to, or farthest from main memory) this PCI-to-PCI bridge. This value is

	stored in the device tree as the second datum in the PCI-to-PCI Bridge's bus-range property.
Secondary Latency Timer	Written by system Open Firmware. Set to 0x20 for all devices, which corresponds to 32 PCI clock intervals.
I/O Base	Written by system Open Firmware. If devices found behind the PCI-to-PCI bridge require I/O space address allocation, this byte-wide register is written with the appropriate values corresponding to the base of I/O space located behind the PCI-to-PCI bridge. See the PCI-to-PCI bridge architecture specification (described in "PCI Special Interest Group" on (page 28)) for details on this register. If no I/O space is requested behind the PCI-to-PCI Bridge, the I/O Base Register is written with a value greater than the I/O Limit value, thereby disabling any decoding of I/O space behind a PCI-to-PCI bridge.
I/O Limit	Written by system Open Firmware. If devices found behind the PCI-to-PCI bridge require I/O space address allocation, this byte-wide register is written with the appropriate values corresponding to the base of I/O space plus the amount of space required located behind the PCI-to-PCI bridge. See the PCI-to-PCI bridge architecture specification for details on this register. If no I/O space is requested behind the PCI-to-PCI Bridge, the I/O Base Register is written with a value greater than the I/O Limit value, thereby disabling any decoding of I/O space behind a PCI-to-PCI bridge.
Secondary Status	Read by Open Firmware. Bit specifics: Bits 10-9, DEVSEL speed, the value stored in the node's `devsel-speed` property. Bit 7, Fast Back to Back capable, a value set for each PCI device. If the value is non-zero, the property `fast-back-to-back` is created for the node (see Command register explanation of Fast Back to Back Enable bit). No specific action is taken based on values of the remaining bits in the Secondary Status Register.
Memory Base	Written by system Open Firmware. If devices found behind the PCI-to-PCI bridge require memory space address allocation, this byte-wide register is written with the values

	corresponding to the base of memory space located behind the PCI-to-PCI bridge. See the PCI-to-PCI bridge architecture specification for details on this register. If no memory space is requested behind the PCI-to-PCI bridge, the Memory Base Register is written with a value greater than the Memory Limit value, thereby disabling any decoding of memory space behind a PCI-to-PCI bridge.
Memory Limit	Written by system Open Firmware. If devices found behind the PCI-to-PCI bridge require memory space address allocation, this byte-wide register is written with values corresponding to the base of memory space plus the amount of space required behind the PCI-to-PCI bridge. See the PCI-to-PCI bridge architecture specification for details on this register. If no memory space is requested behind the PCI-to-PCI bridge, the Memory Base Register is written with a value greater than the Memory Limit value, thereby disabling any decoding of memory space behind a PCI-to-PCI bridge.
Prefetchable Memory Base	Written by system Open Firmware. All memory space allocated behind a PCI-to-PCI bridge in PCI Power Macintosh systems is defined as non-prefetchable. Therefore, the Prefetchable Memory Base register is always written with a value that is greater than the Prefetchable Memory Limit value. This disables any decoding of Prefetchable Memory behind a PCI-to-PCI bridge.
Prefetchable Memory Limit	Written by system Open Firmware. All memory space allocated behind a PCI-to-PCI bridge is defined as non-prefetchable. Therefore, the Prefetchable Memory Base register is always written with a value that is greater than the Prefetchable Memory Limit value. This disables any decoding of Prefetchable Memory behind a PCI-to-PCI bridge.
Prefetchable Base Upper 32 bits	Written by system Open Firmware with all 0s, because the PCI Power Macintosh computers have a 32-bit address space.
Prefetchable Limit Upper 32 bits	Written by system Open Firmware with all 0s, because the

	PCI Power Macintosh computers have a 32-bit address space.
I/O Base Upper 16 bits	Written by system Open Firmware with all 0s, because the PCI Power Macintosh computers utilize a 16-bit I/O address space behind PCI-to-PCI bridges.
I/O Limit Upper 16 bits	Written by system Open Firmware with all 0s, because the PCI Power Macintosh computers utilize a 16-bit I/O address space behind PCI-to-PCI bridges.
Expansion ROM Base Register	Open Firmware takes no action with this register. It is assumed that PCI-to-PCI bridges have no FCode in their ROMs.
Interrupt Line	No action taken on this register. The value in this register has no meaning for the Power Macintosh computers, because interrupts are OR-combined per slot in hardware, creating a unique interrupt for each PCI card accessible to the system interrupt controller. No useful information for Power Macintosh driver writers exists in this register.
Interrupt Pin	Read by system Open Firmware. If the value is nonzero, it appears in the property `interrupts`. It has no meaning for Power Macintosh, for the reasons given in the preceding paragraph.
Bridge Control	Written by system Open Firmware. Bit specifics: Bit 7, Fast Back to Back Enable, is written 1 if all PCI devices on the secondary side of the PCI-to-PCI bridge are Fast Back to Back capable (if all devices have a fast-back-to-back property stored in their device node); otherwise, it is written 0. Bit 6, Secondary Bus Reset, is written 0 so as not to cause a separate reset on the secondary bus from the regular PCI hardware reset, which is passed automatically by the PCI-to-PCI bridge hardware. Bit 5, Master Abort Mode, is written 0 so that all Master Aborts on the Secondary bus return all Fs on read actions. Bit 4, Reserved.

Bit 3, VGA Enable, is written 0, which disallows the forwarding of VGA hard decoding addresses to the secondary bus.

Bit 2, ISA Enable, is written 1, which blocks forwarding of traditional hard-decoded addresses (top 768 bytes for each 1K block of I/O space) from the primary to the secondary PCI bus.

Bit 1, SERR# Enable, is written 0, because the Power Macintosh system doesn't respond to SERR signals.

Bit 0, Parity Error Response, is written 0.

C H A P T E R 5

PCI Open Firmware Drivers

As explained in Chapter 4, “Startup and System Configuration,” PCI expansion cards in Power Macintosh computers may need to operate during the Open Firmware startup process, before any operating system is present. The drivers for such cards are called *Open Firmware drivers.* Other drivers, called *run-time drivers,* are used only after an operating system has been loaded and has taken control of the main processor. Read “Open Firmware FCode Options” (page 87), for help in deciding whether or not your PCI card needs an Open Firmware driver.

This chapter discusses the general technical requirements for Open Firmware drivers for PCI devices—drivers that are used with the Open Firmware startup process. Run-time drivers for PCI devices used with Mac OS and other operating systems are discussed in Part 3, “Native Device Drivers.”

General Requirements

Any Open Firmware driver must be stored in a PCI card’s expansion ROM so that the Macintosh firmware can load and run the driver prior to the invocation of the disk-based operating system. Open Firmware drivers are written in FCode. For further information about FCode, see *Writing FCode Programs for PCI*. This book is listed in “Other Publications” (page 26).

Other general requirements for Open Firmware drivers include the following:

- They must be able to acquire any software resources they need from the PCI card’s expansion ROM or from the Macintosh firmware. For example, a display card must be able to access a font in the expansion ROM if it is required to write characters on the screen during startup.
- The card hardware may not address system space below 1 MB. In Power Macintosh computers, PCI cards that request space below 1 MB in a `reg` property will not receive a corresponding `assigned-addresses` entry.
- PCI expansion cards and their drivers should avoid hard address decoding, as discussed in “Hard Decoding Device Address Space” (page 55).

Driver Interfaces

Open Firmware driver code typically supports two interfaces:

- a hardware interface, through which the driver controls its associated device
- a client interface, through which the driver cooperates with an operating system

Discussion of the hardware interface for Open Firmware driver code is beyond the scope of this book; it is assumed that the relation between a driver and its associated hardware is entirely controlled by the internal design of the PCI expansion card.

This book also does not try to discuss the general client interface for Open Firmware drivers, which is of interest primarily to engineers designing an operating system. For details about the specific client interface between drivers and Mac OS, see Part 3, "Native Device Drivers" (page 137).

The next section discusses how PCI card expansion ROMs export properties to the Open Firmware device tree. This process lets the card's Open Firmware drivers (if any) work with the Power Macintosh firmware during the computer's startup process, before an operating system is present.

IMPORTANT

To participate in the boot process, PCI expansion cards installed in Macintosh computers that support the NewWorld system architecture must include Open Firmware drivers in the PCI card expansion ROM. ▲

Open Firmware Driver Properties

When the Open Firmware startup process finds a PCI expansion card, it looks in the card's expansion ROM for an Open Firmware signature and succeeding FCode. When it finds FCode, the Open Firmware startup process loads it into RAM and interprets and executes it. The code must fill in the part of the device tree applicable to its device node; it must also create property nodes required by the startup firmware and by any operating system that may use the driver in the future.

The standard property nodes for PCI devices working with the Open Firmware startup process are defined in *PCI Bus Binding to IEEE 1275-1994.* For information about obtaining this document see the note under "Institute of Electrical and Electronic Engineers" (page 27).

The call interface to PCI Open Firmware drivers and the data format for the Open Firmware signature are defined in IEEE Standard 1275. This book is listed in "Supplementary Documents" (page 26).

Standard device properties for PCI expansion cards and run-time drivers used with Mac OS are listed in Table 10-1 (page 322). The same properties are used with boot devices and Open Firmware drivers for Power Macintosh computers. Other properties, described in IEEE Standard 1275, may be required if a PCI card is to support operating systems other than Mac OS, or be compatible with computers besides Power Macintosh.

Open Firmware User Interface

Open Firmware is the process that controls the microprocessor after hardware initialization and diagnostics are performed, but before the main operating system is passed control. Open Firmware is responsible for, among other things, building the device tree and probing the expansion slots for I/O devices. Open Firmware queries PCI devices for address space requirements and dynamically assigns the needed address space to each device. It is during this probing process that each device and ASIC on the logic board is given a node in the device tree. Hardware and software engineers can use the Open Firmware user interface to debug their device and driver, respectively. See Technote 1044, Understanding PCI Expansion Choices for Mac OS 8, Part III in the Open Firmware Technote Series, for details about properties and methods for various devices. You must be able to traverse the device tree to get to your device node and then to edit and debug that node.

The Macintosh implementation of Open Firmware includes the user interface described in IEEE Standard 1275. The user interface provides an interactive terminal environment that is useful in viewing and manipulating Open Firmware data structures and other system-level resources, such as memory and device registers, in the absence of a running operating system. On Power

Macintosh PCI computers that don't support the NewWorld architecture, the default implementation operates from a remote terminal connected by a serial communication link to the modem port of the target PCI-based Power Macintosh computer. On Power Macintosh computers that support the NewWorld architecture, the default implementation operates in a one machine mode, and can be set to remote serial mode. The serial link's default settings are as follows:

Open Firmware version 1.0 and 2.0 use 38400 baud
Open Firmware version 3.0 uses 57600 baud
No parity
8 data bits
1 stop bit
XON/XOFF handshake
ANSI/VT102 terminal protocol

In addition to the remote terminal method of accessing Open Firmware, Power Macintosh G3 computers support invocation of the user interface on the current machine without the need for a remote serial connection. See "Invoking the User Interface On the Current Machine" (page 118), for additional information.

Invoking the User Interface Via Remote Connection

To enter the Open Firmware user interface, restart the target Power Macintosh computer while you immediately and simultaneously press the Command, Option, O, and F keys on its keyboard. (O and F represent Open Firmware.) Release the keys after you see the Open Firmware prompt on the screen or in the case of pre-NewWorld computers, the remote terminal. It should look similar to the following example:

```
Open Firmware, 3.1.0
To continue booting the MacOS type:
BYE<return>
To continue booting from the default boot device type:
BOOT<return>
ok
0 >
```

The Open Firmware version number varies upon the target. Macintosh computers that support the New World architecture have Open Firmware version 3.0.x. or greater. Older PCI-based Power Macintosh computers include Open Firmware versions up to 2.0.x.

The "`ok`" means the Open Firmware Forth interpreter is waiting for keyboard input. The `0` indicates the top of the stack.

If you see the Mac OS boot screen, "Welcome to Macintosh," on the target computer, you may have failed to press the keys quickly enough and should try again.

The Command, Option, O, and F key action just described causes the Macintosh startup firmware to enter the Open Firmware user interface at the point just before initiating an operating system startup process. At this point all FCode that was present on PCI cards has been executed and the `assigned-addresses` and other standard properties have been added to the device tree. When the user interface is invoked, it sends a bell character and a string identifying Open Firmware and its version number to the remote terminal. It then awaits input from the terminal. The default routes for both output and input devices are through the serial terminal connection.

To move from two machine to one machine mode during an individual session, enter the following redirection words:

```
0 > " pci2/@f" output \ the path must point to your display node
0 > " kbd" input \ the alias kdb could be " keyboard"
```

Note: `" kbd"` is an alias supported on some the machines, others may use `" keyboard"` instead. But there is no standard alias for display screen. The redirection of output is specific to pre-New World machines with Open Firmware version 2.0 or later. You will have to decide what is the display for the target machine and use that path name.

After the output is directed to the target machine, what you enter at the host for the second word (i.e., input) does not appear on the host display, but on the target display. You can now enter your session on the target machine until you restart your target. Once you restart, your input and output capabilities will again be at the host. To make the redirection change permanent, use the `printenv` and `setenv` words. Listing 5-1 shows an example of the display output produced after entering the `printenv` word and pressing the retun key.

The output shown Listing 5-1 is representative of what you would see if you entered `printenv` on a current Power Macintosh G3 computer. There are two columns displaying values for the configuration variables; the left column displays current setting; the right column displays the default setting. You must change the environmental variables called `input-device` and `output-device` to contain the path name to your keyboard and display, respectively. You use the `setenv` word followed by the variable name and a new value to change the variables. Then, when you restart your target, you will always be in one machine mode. Of course, since these variables are stored in NVRAM, you can reset them to the default behavior using the Option-Command-P-R keys upon restart.

If the Open Firmware configuration variable `auto-boot?` is set to `false`, the Macintosh startup firmware enters the user interface automatically after subsequent system restarts. This makes the Command-Option-O-F key combination unnecessary.

Listing 5-1 Example of printenv output

```
0 > printenv
------------- Partition: common -------- Signature: 0x70 --------------
little-endian?          false               false
real-mode?              false               false
auto-boot?              false               true
diag-switch?            false               false
fcode-debug?            false               false
oem-banner?             false               false
oem-logo?               false               false
use-nvramrc?            false               false
use-generic?            false               false
default-mac-address?    false               false
real-base               -1                  -1
real-size               -1                  -1
load-base               0x800000            0x800000
virt-base               -1                  -1
virt-size               -1                  -1
pci-probe-mask          -1                  -1
screen-#columns         100                 100
screen-#rows            40                  40
selftest-#megs          0                   0
boot-device             hd:5,\\:tbxi        hd:5,\\:tbxi
boot-file
boot-screen
console-screen
```

```
diag-device             floppy              floppy
diag-file               diags               diags
input-device            keyboard            keyboard
output-device           screen              screen
mouse-device            mouse               mouse
oem-banner
oem-logo
nvramrc
boot-command            mac-boot            mac-boot
forced-boot
fw-scsicfg
fw-boot-path
default-client-ip
default-server-ip
default-gateway-ip
default-subnet-mask
default-router-ip
boot-script
aapl,pci                Use PRINT-AAPL,PCI to view
ASVP                    30313036 30333030 31373030
 ok
0 >
```

Invoking the User Interface On the Current Machine

Starting with the PowerBook 3400 and desktop Power Macintosh G3 computers, you can enter the Open Firmware user interface on the current machine via the keyboard and display without setting up a remote serial connection with a second computer. The process for invoking the Open Firmware user interface on the current machine is the same.

Startup the computer and simultaneously press the Command, Option, O, and F keys on its keyboard. Release the keys after you hear the boot sound from the computer and see the Open Firmware prompt. Power Macintosh G3 computers equipped with external serial ports also support remote connection to the Open Firmware user interface, as described in “Invoking the User Interface Via Remote Connection” (page 115).

User Interface Environment

The user interface operates as an interactive Forth environment, with necessary omissions and additions as appropriate to Open Firmware. The interface should be used to develop and debug the Forth source code that will eventually be converted into FCode and stored in a PCI card's expansion ROM. To create FCode, which is a tokenized representation of the Forth source, you must use an FCode tokenizer. Apple provides the standalone tokenizer in the PCI driver development kit (DDK). Special tokenizer words automatically generate a ROM image with the correct signatures and formats for a PCI card expansion ROM with FCode.

The Open Firmware interface interprets common Forth words as well as user defined words. The following sections list some of the common Forth commands that are available for Open Firmware driver development and debugging. The Forth language interpreter and Open Firmware implementation details are well defined in the documentation listed in "Other Publications" (page 26). This chapter provides only a brief introduction the features available in the Open Firmware environment.

Note
While testing your forth code in the Open Firmware interface, you may get stuck in a situation that causes an infinite loop. You can get out of this situation by shutting down Open Firmware with a press of the Power key on the keyboard.

Open Firmware Forth Language Symbols

Table 5-1 lists a few of the common symbols used in Open Firmware Forth code. See "Open Firmware Forth Usage Examples" (page 126), for examples that use some of the symbols listed here.

Table 5-1 Open Firmware common symbols

Symbol	Meaning	Usage
`@`	fetch	get the contents of address
`!`	store	store data at address
`c`	char	8 bit datum
`w`	word	6 bit datum
`l`	long	32 bit datum
`adr`		32 bit address

Defining Forth Words

The : (colon) and ";" (semicolon) are used when defining forth words (word in this case is not the same as the term word defined in Table 5-1). To start a Forth word definition use the colon <:> symbol, to end a word definition, use the <;> semicolon. Here is an example of a word definition:

```
: HI ." Hello New World";
```

The example also uses the word `."` (dot quote) within the HI colon definintion. The word `."` (dot quote) tells the interpreter to collect the characters that follow it until reaching a closing `"` (quote) character. Words are often combinations of other words that provide the functions needed to accomplish the required result. In this case, the definition of the HI specifies that the interpreter is to collect the text, *Hello New World*. If you enter the above definition, press the Return key, and then type `HI` and press the Return key again, the characters `Hello New World` followed by the prompt are displayed.

The following section lists, and briefly defines, commonly used Forth commands.

Forth Commands (Words)

The tables in this section list some of the common Forth commands that you may find useful while developing and debugging PCI card Open Firmware FCode drivers.

Table 5-2 lists some of the common Forth language number base commands.

Table 5-2 Forth number base commands

Command	**Definition**
`decimal` (--)	Set number base to ten
`d# number` (-- *n*)	Interpret the next number as decimal
`hex` (--)	Set number base to 16
`h# number` (-- *n*)	Interpret the next number as hex
`.d` (*n* --)	Display *n* in decimal without changing base
`.h` (*n* --)	Display *n* in hex without changing base
`.u` (*n* --)	Display unsigned number

Table 5-3 lists common Forth language commands used for operations on the stack.

Table 5-3 Forth stack commands

Command	**Definition**
`.` (*n* --)	Display contents of top of stack in current base
`.s` (--)	Display contents of the stack
`clear`	Empty the contents of the stack
`drop` (*n* --)	Remove the top item on the stack
`dup` (*n* -- *nn*)	Copy the top item on the stack

Table 5-3 Forth stack commands (continued)

Command	Definition
`over` (*n1 n2* -- *n1 n2 n1*)	Copy top 2nd item to the top of the stack
`rot` (*n1 n2 n3* -- *n2 n3 n1*)	Rotate top 3 items on the stack
`swap` (*n1 n2* -- *n2 n1*)	Swap top 2 items

Table 5-4 lists common Forth language arithmetic commands.

Table 5-4 Forth arithmetic commands

Command	Definition
`*` (*n1 n2* -- *n3*)	Multiply
`+` (*n1 n2* -- *n3*)	Add
`-` (*n1 n2* -- *n3*)	Subtract
`/` (*n1 n2* -- *n3*)	Divide
`<<` (*n1 n2* -- *n3*)	Shift left *n*1 by *n*2 bits
`>>` (*n1 n2* -- *n3*)	Shift right *n*1 by *n*2 bits
`and` (*n1 n2* -- *n3*)	Bitwise AND operation
`mod` (*n1 n2* -- *n3*)	Remainder of *n*1/*n*2
`not` (*n1* -- *n2*)	Bitwise inversion
`or` (*n1 n2* -- *n3*)	Bitwise OR
`xor` (*n1 n2* -- *n3*)	Bitwise XOR

Table 5-5 lists common Forth language conditional commands.

Table 5-5 Forth conditional commands

Command	Definition
`if` (*?* --)	Execute the following if the flag is true
`then` (--)	Terminate IF
`else` (--)	Execute the following if the flag is false

Table 5-6 lists common Forth language commands for loop operations.

Table 5-6 Forth commands for loop operations

Command	Definition
`do` (*end start* --)	Begin a do loop (ex: 10 0 do i . loop)
`loop` (--)	End a do loop construct
`+loop` (n --)	End a do loop construct adding n each pass
`begin` (--)	Start non indexed loop
`again` (--)	End a non indexed loop construct
`until` (? --)	End a begin loop when flag is true
`begin` (--)	Start non-indexed loop
`while` (? --)	End a non-indexed loop construct
`repeat` (--)	End of loop construct

Table 5-7 lists common Forth language keyboard commands.

Table 5-7 Forth commands for keyboard operations

Command	**Definition**
`key?` (-- *n*)	Returns true/false if key has been typed
`key` (-- *n*)	Read a key from keyboard
`cr` (--)	Go to next line on display

Table 5-8 lists common Forth language commands for memory mapping operations.

Table 5-8 Forth Memory mapping commands

Command	**Definition**
`do-map` (*phys virt size mode* --)	Map a region of 32 bit physical address space to virtual memory space. Mode = 0 for memory, 0x28 for I/O. Space must be mapped before access is possible.
`do-unmap` (*virt size* --)	Unmap previously mapped space

Open Firmware User Interface Commands and Examples

A short list of commands available through the Open Firmware user interface is shown in Table 5-9. Included within some of the command descriptions are examples that further illustrate command usage. Note that several of the commands are combinations of commands that can be used separately. The

Open Firmware IEEE 1275 specification defines all of the commands and keywords.

Table 5-9 Commonly used Open Firmware user interface commands

Commands	**Definition**
`assign-addresses`	Emulates the regular Open Firmware startup process of querying the system for resource requirements and adding an `assigned-addresses` property to the node that is the current package.
`boot`	Performs the startup process, using the currently chosen device.
`ls`	List the children of the current node.
`dev / ls`	Selects the root node and lists its children recursively, effectively dumping a name view of the device tree. Press Control-S to stop the text from scrolling off the screen. Restart the scrolling with Control-Q.
`devalias`	Show currently defined device aliases
`dev` <*dev pathname*>	Selects the specified device in pathname and makes it the current node.
`dev /bandit/gc .properties`	Selects `gc` (the node representing the Bandit ASIC, which controls many Macintosh I/O features) as the active package and displays its properties. Bandit is used in the first PCI-based Power Macintosh models but may not be present in future models. For an illustration of its position in the device tree, see Listing 10-1 (page 284).
show-devs <*dev pathname*>	Show devices beneath the device node specified in pathname.
`dl`	Sets the terminal emulator for downloading Forth code to RAM. Press Control-D to end the downloading process.
`dump-device-tree`	Lists properties and methods of all the device tree nodes.
`FFC00000 100 dump`	Dumps 0x100 bytes from virtual address 0xFFC00000, if that address is currently mapped in.
`init-nvram`	Resets data in NVRAM to default values.

Table 5-9 Commonly used Open Firmware user interface commands (continued)

Commands	**Definition**
`properties`	Show the names and values of the current device node properties.
`make-properties`	Emulates the regular Open Firmware startup process of querying the device's configuration space and adding the standard PCI properties to the node that is the current package.
`printenv`	Shows the current environment and default settings of Open Firmware configuration variables stored in NVRAM.
`setenv` *param val*	Sets the current environment parameter to the specified value. The value set by `setenv` persist across restarts.
`setenv auto-boot? false`	Sets the environment variable `auto-boot?` stored in NVRAM to `false`. This causes the computer to invoke the Open Firmware user interface automatically after subsequent restarts.
`shut-down`	Powers down the computer.
`pwd`	Displays the pathname of the current package.
`reset-all`	Resets the target computer.
`see` *word*	Displays the Forth source code for the word entered.
`words`	Lists variables, constants, and methods of the active package (as in Forth, but in the scope of the current package only).

Open Firmware Forth Usage Examples

Here are a few short examples that show what can be done with the Forth language while in the Open Firmware interface.

The following example maps the physical address 8000 to virtual memory space for 0x1000 bytes for access.

```
8000 8000 1000 0 do-map
```

The next example dumps the contents of address 8000 continuously until any key press.

```
begin 8000 l@ . key? until
```

The next example stores 0x12345678 at location 8000.

```
12345678 8000 l!
```

The next example dumps 16 longs starting from location 8000.

```
8000 40 bounds do cr i u. i l@ . 4 +loop
```

The next example writes 0xaa55 to location 8000 until keypress.

```
begin aa55 8000 l! key? until
```

Terminal Emulation in Graphics Drivers

For details of Open Firmware driver design for most standard boot devices, including Open Firmware graphics drivers, see IEEE Standard 1275 and *Writing FCode Programs.* These books are listed in “Other Publications” (page 26).

Besides their generic requirements, Open Firmware drivers for PCI graphics cards in Power Macintosh computers must provide terminal emulation support. IEEE Standard 1275 defines the behavior of a terminal emulator support package, including the implementation of certain escape sequences defined by ANSI Standard X3.64. The Macintosh package, described here, conforms to ISO Standard 6429-1983. The Macintosh implementation of Open Firmware for PowerPC supports additional graphic renditions, through Select Graphic Rendition (SGR) escape sequences, beyond those specified in the Open Firmware standard.

For the Macintosh terminal emulation extensions to be used, the FCode device driver for a display device (a device whose `device_type` property has the value `display`) must initialize the first 16 entries of its color table to appropriate values, as described below. These values assume that the color is represented by the low-order 3 bits of the color index and that the bit corresponding to a value of 8 represents the intensity. The ISO Standard 6429-1983 provides parameter values to control the color of foreground (30–37) and background (40–47) independently. The intensity is set separately (1–2), and must be issued before the color control; 1 -> color, 2 -> color+8.

In the Macintosh terminal emulator, there are current background and foreground colors whose values range from 0 through 15, corresponding to the first 16 entries of the color table. In positive image mode, pixels corresponding to a font or logo bit set to a value of 1 are set to the foreground color; pixels corresponding to a font or logo bit cleared to 0 are set to the background color. When in negative image mode, the roles of foreground and background are reversed.

The default rendition is positive image mode, with background set to 15 and the foreground set to 0, thus producing black characters on a bright white background.

Table 5-10 lists the effects of executing SGR escape sequences with various parameters.

Table 5-10 SGR escape sequence parameters

Parameter	Interpretation
0	Default rendition
1	Bold (increased intensity)
2	Faint (decreased intensity)
7	Negative image
27	Positive image
30	Black foreground
31	Red foreground
32	Green foreground
33	Yellow foreground
34	Blue foreground
35	Magenta foreground
36	Cyan foreground
37	White foreground
40	Black background
41	Red background

Table 5-10 SGR escape sequence parameters (continued)

Parameter	Interpretation
42	Green background
43	Yellow background
44	Blue background
45	Magenta background
46	Cyan background
47	White background

The next sections define the additional behavior of display devices for Open Firmware implementations that support the terminal emulator extensions.

Color Table Initialization

The core specification of Open Firmware defines a terminal emulation support package that does not include support for colors. The Macintosh Open Firmware implementation supports additional SGR parameters to allow client programs to display characters and logos in a 16-color model.

For this expanded terminal emulation support to work, Open Firmware device drivers for display devices must initialize the first 16 entries of their color table to values defined in Table 5-11, where values are defined in terms of the fraction of full saturation required for each of the primary red-green-blue (RGB) colors.

Table 5-11 Color table values

Index	Red	Green	Blue	Color
0	0	0	0	Black
1	0	0	2/3	Blue
2	0	2/3	0	Green
3	0	2/3	2/3	Cyan
4	2/3	0	0	Red

Table 5-11 Color table values (continued)

Index	Red	Green	Blue	Color
5	2/3	0	2/3	Magenta
6	2/3	1/3	0	Brown
7	2/3	2/3	2/3	White
8	1/3	1/3	1/3	Gray
9	1/3	1/3	1	Light blue
10	1/3	1	1/3	Light green
11	1/3	1	1	Light cyan
12	1	1/3	1/3	Light red
13	1	1/3	1	Light magenta
14	1	1	1/3	Light yellow
15	1	1	1	Bright white

Display Device Standard Properties

In addition to the standard properties defined by Open Firmware for display devices, the following device properties, encoded as with `encode-int`, must be supported:

`width`	Visible width of the display, in pixels.
`height`	Visible height of the display, in pixels.
`linebytes`	Address offset between a pixel on one scan line and the same horizontal pixel position on the next scan line.
`depth`	Number of bits in each pixel.

Display Device Standard Methods

This section defines additional methods that display devices should implement to be compliant with the Macintosh terminal emulator extensions. These methods assume that the device supports at least 16 colors using the RGB color model and that a color lookup table (CLUT) exists that can be read and written to. The model assumes 8-bit values for each of the RGB components of the colors, where 0x00 implies no color and 0xFF indicates full saturation of the component. If fewer bits are available, the corresponding entries should be scaled appropriately.

Individual color entries are specified by their RGB values, using 8 bits for each. Each color is represented by an index. The index values for the 16-color extension are in the range 0 through 15; however, most display hardware will support at least 256 colors.

The following methods allow access to the CLUT from client programs, as well as the user interface described in the next section.

`set-colors ( adr index #indices -- )`

Allows setting a number of consecutive colors, starting at `index`, for `#indices` colors. The `adr` parameter is the address of a table of packed RGB components.

`get-colors ( adr index #indices -- )`

Allows reading a number of consecutive colors, starting at `index`, for `#indices` colors. The `adr` parameter is the address of a table that will be filled in with packed RGB components.

`color! ( r g b index -- )`

Allows setting a single color value, specified by `index`. The `r`, `g`, and `b` parameters are values to be placed into the red, green, and blue components, respectively.

`color@ ( index -- r g b )`

Allows reading the color components of a single color value, specified by `index`. The `r`, `g`, and `b` parameters are the values of the red, green, and blue components, respectively.

Sample FCode Drivers

Listing 5-2 shows an example of minimal FCode support for a PCI SCSI card. The FCode in the example provides identifying information in its device node and creates a property that contains the run-time driver to be loaded into the Macintosh system heap by the Expansion Bus Manager.

Listing 5-2 Minimal PCI SCSI card FCode support

```
/  push arguments on the stack for pci-header:
/  *** THESE MUST MATCH THE CONFIG REGISTERS FOR YOUR   ***
/  *** FCODE TO BE RECOGNIZED BY OPEN FIRMWARE          ***
/  vendor #, device #, class-code = SCSI bus controller

/                       vendorID        deviceID        classCode
    tokenizer[ hex      1000            0003            010000  ]tokenizer

/ generate proper PCI image header
    pci-header

/ generate proper FCode header (within PCI image)
    fcode-version2
    hex

/ encode name property
/ refer to IEEE 1275-1994, page 162
/ refer to PCI bus binding to IEEE 1275-1994 Revision 2.0,
/ Section 2.5 FCode Evaluation Semantics, page 9, line 36
    "AAPL,scsi" encode-string " name" property

/ encode compatibility property
/ refer to IEEE 1275-1995, page 127
/ refer to Open Firmware Recommended Pratice, Generic Names, version 1.4,
/ Section 3 Generic Names, page 5, Guidline 2
    "APPL,SYN53C875J" encode-string " compatibility" property

/ encode device_type property
/ Ref : IEEE Std 1275-1994 : page 208
" scsi-2" encode-string " device_type" property

/ encode reg property
/ Ref : IEEE Std 1275-1994 : page 174
/ Ref : Writting FCode Programs For PCI : page 84

/ generate a "reg" property which lists our configuration space at the start
/ of the assigned space, with 0 size (as required by the PCI Binding
/ Supplement)
/ PCI Configuration Space entry of the reg property
my-address my-space encode-phys                                 \ configuration space
0 encode-int        encode+     0 encode-int        encode+     \ size ( 0 )

/ Base Address Zero entry of the reg property
my-address my-space 01000010 or encode-phys                 encode+     \ I/O space
0 encode-int        encode+     100 encode-int              encode+     \ size ( 256 )

/ Base Address One entry of the reg property
my-address my-space 02000014 or encode-phys                 encode+     \ memory space
0 encode-int        encode+     100 encode-int              encode+     \ size ( 256 )

/ Base Address Two entry of the reg property
my-address my-space 02000018 or encode-phys                 encode+     \ memory space
0 encode-int        encode+     1000 encode-int             encode+     \ size ( 4K )

/ PCI Expansion ROM entry of the reg property
my-address my-space 02000030 or encode-phys                 encode+     \ expansion rom
0 encode-int        encode+     10000 encode-int            encode+     \ size ( 64K )

" reg" property

/ "power-consumption" property which lists standby and full-on power
/ consumtion for various power rails in microwatts would go here; if you
/ don't create this property, Open Firmware creates one by filling in the
/ "unspecified" rail entries from the PRSNT pins (if you don't know the
/ power consumption values, you can fill the "unspecified" entries with
/ zeros)

0 encode-int 0 encode-int encode+                                       / "unspecified"
d# 7500000 encode-int d# 7500000 encode-int encode+ encode+             / +5V
0 encode-int 0 encode-int encode+ encode+                               / +3V
d# 8100000 encode-int d# 8100000 encode-int encode+ encode+             / I/O power
/ remaining entries are 0 and can be omitted
0 encode-int 0 encode-int encode+ encode+                               / reserved

"power-consumption" property

/ the following properties will automatically be generated for this card by
/ Open Firmware from the PCI Configuration Space Header.
/ Ref : PCI Bus Binding to : IEEE Std 1275-1994 Revision 2.0, Section 2.5,
/ FCode Evaluation Semantics : page 8 : lines 14 - 57.
/ "has-fcode"
/ "vendor-id"
/ "device-id"
/ "revision-id"
/ "class-code"
/ "interrupts"
/ "min-grant" - unless header is type 01h
/ "max-latency" - unless header is type 01h
/ "devsel-speed"
/ "fast-back-to-back"
/ "subsystem-id"
/ "cache-line-size"
/ "66MHz-capable"
/ "udf-supported"
/  "assigned-addresses"

/ there is enough information for the runtime driver to be able to locate the card,
/ however a complete FCode implementation would also provide boot-time I/O services

/ include an image of the runtime driver, and have it assigned as the value of a
/ property that the Mac OS will read at startup

fcode-end           / end FCode header
pci-end             / end PCI header
```

Listing 5-3 Minimal network FCode driver support

```
/ Network drivers should support the
/ TFTP Booting Extension Recommened Pratice
/ [promiscous], [speed= n], [duplex= mode], [bootp], [siaddr], [filename],
/ [ciaddr], [giaddr], [bootp-retries], [tftp-retries]
/ See the PCI DDK 3.0 for a complete Network driver example

" ethernet" device-name
" network" device-type
" ethernet" encode-string " network-type" property
" network" encode-string " removable" property
" net" encode-string " category" property
```

Listing 5-4 Minimal display FCode driver support

```
/ See the PCI DDK 3.0 for a complete display driver example
" display" device-type                      / defines the device type as display

: my-open ( -- )
    ... initialize device                   / your device initialization
    ... monitor sense                       / monitor sense code
    ;
: my-close ( -- )
    ...             / close code goes here
    ;

[‘] my-open is-install
    / You could put fcode in this area that puts up boot time options or a logo
    / in a terminal emulator.
    / here is a prototype example
    : open ( -- true )
        ...
        my-open
        ;
    : write ( adr len -- actual )
        ...
        ;
    : draw-logo ( l# a w h -- )
        ...
        ;
    : restore
    ;
[’] my-close is-remove
    : close ( -- )
        my-close
        ;
/ If you want to draw with color, you should support the
/ Open Firmware 8-bit graphics extensions recommended pratice. See also,
/ “Terminal Emulation in Graphics Drivers” (page 127).
/ The basic requirements are:
        / draw-rectangle ( adr x y w h -- )
        / fill-rectangle ( index x y w h -- )
        / read-rectangle ( adr x y w h -- )
        / color! ( r g b index -- )
        / color@ ( index -- r g b )
        / set-colors ( adr index #indices -- )
        / get-colors ( adr index #indices -- )
```

P A R T   T H R E E

Native Device Drivers

This part of *Designing PCI Cards and Drivers for Power Macintosh Computers* tells you how to design and write run-time native device drivers that support the PCI-bus compatible Power Macintosh architecture. These drivers are called *native* because they are written for execution by the native instruction set of the PowerPC microprocessor. This part consists of the following chapters:

- Chapter 6, "Native Driver Overview," presents the general concepts and framework applicable to native drivers for PowerPC Macintosh computers.
- Chapter 7, "Finding, Initializing, and Replacing Drivers," describes how the Macintosh hardware and software architecture determine how drivers and devices are matched, and discusses what driver and card designers can do to improve the compatibility of their products on the Macintosh platform.
- Chapter 8, "Writing Native Drivers," gives you details of native driver design and coding, including how to use services provided by the Mac OS Driver Loader Library.
- Chapter 9, "Driver Loader Library," describes the Driver Loader Library (DLL), a CFM shared-library extension to the Macintosh Device Manager.
- Chapter 10, "Name Registry," describes the Mac OS data structure that stores device information extracted from the PCI device tree.
- Chapter 11, "Driver Services Library," details the general support that Mac OS provides for device drivers, including interrupt and timing services.
- Chapter 12, "Expansion Bus Manager," discusses a collection of PCI bus-specific system services available to native device drivers.
- Chapter 13, "Graphics Drivers," describes the calls serviced by typical display drivers.
- Chapter 14, "Network Drivers," describes how to construct a sample network driver.
- Chapter 15, "SCSI Drivers," describes how to construct a sample native SCSI Interface Module (SIM) compatible with SCSI Manager 4.3.

C H A P T E R 6

Native Driver Overview

This chapter presents an overview of the native driver environment and services, or *I/O architecture,* available in the Mac OS for Power Macintosh computers that include a PCI bus. It covers concepts and terminology that are introduced with this I/O architecture. It also provides a high-level summary of the new driver interfaces, packaging, and support. The discussion in this chapter applies to run-time drivers, which run after the system startup steps detailed in Chapter 4, "Startup and System Configuration."

The previous Macintosh I/O architecture was based on resources of type `'DRVR'` and their associated system software, including the Device Manager. Mac OS now supports a more general concept of driver software. In the new I/O architecture, a **driver** is any PowerPC native code that controls a physical or virtual device. This definition includes resources of type `'ndrv'` but excludes resources of type `'DRVR'`, protocol modules, control panels, resources of type `'INIT'`, and application code.

Native device drivers are now isolated from application-level interfaces and services; in particular, main driver code must run without access to the Macintosh Toolbox. This concept is discussed further in "Separation of Application and Driver Services" (page 144).

To understand this chapter, you should have some experience developing drivers or similar software designed to work with Mac OS. For recommended reading material about Macintosh technology, see the documents listed in "Supplementary Documents" (page 26).

Macintosh System Evolution

For their the second generation, Power Macintosh computers are switching from NuBus to the more standard PCI bus. This change means that many useful new PCI-based peripheral devices will become available for Macintosh computers.

To provide improved I/O support in Mac OS, Apple is introducing a **native I/O framework** that includes a set of driver services and mechanisms separate from those available to previous Macintosh device drivers. The native I/O framework includes these new features:

- native PowerPC execution of all driver code
- support modern types such as PCI bus and Card Bus

- new Device Manager support for concurrent operations
- improved interrupt mechanisms
- driver support services
- a Name Registry

Mac OS provides these features only for native device drivers. Existing drivers written in code for MC68000-family microprocessors (called *68K drivers*) will continue to work as they have in the past, but inclusion of the new I/O framework marks the beginning of the transition of all Mac OS drivers to the native model described in this chapter. The model standardizes Mac OS driver design so that PCI and non-PCI device drivers can be written to a single specification.

Terminology

The following list defines new terms used in the rest of this book:

- **Application programming interface (API):** The API is the rich set of Mac OS services available to application-level software, including the Macintosh Toolbox routines. Drivers do not have access to this set of services.
- **Code Fragment Manager (CFM):** The CFM is the part of Mac OS that loads code fragments into RAM and prepares them for execution. The CFM is described in *Inside Macintosh: PowerPC System Software.*
- **Disk-based driver:** Disk-based drivers are drivers that are stored in the Mac OS file system, in the Extensions folder. Disk-based drivers are CFM fragments in files of type `'ndrv'` with an unknown creator. A disk-based driver may replace a ROM-based driver if it is a newer version. Disk-based drivers are not available during system startup, before the file system is working.
- **Expert:** The code that connects a class or family of devices to the operating system is called an *expert. Low-level experts* and *family experts* are defined below.
- **Family:** A device family is a collection of devices that provide the same kind of I/O functionality. One example of a family is the set of Open Transport devices with their corresponding Open Transport Data Link Provider Interface (DLPI) drivers. Another example is the family of display devices.

- **Family expert:** A family expert, or *high-level expert*, is the code responsible for locating, initializing, and monitoring all entries in the Name Registry that are associated with devices in its family or service type. Hence, a family expert is the device administrator for a family. Family experts run when devices are connected to the system (usually at system startup time), but they are not part of the primary data paths to the devices.
- **Family programming interface (FPI):** An FPI is a set of services used between a family expert and the devices in the expert's family. For example, Open Transport exports the routine `freemsg` as part of its FPI. This routine returns a STREAMS buffer to the general memory pool maintained by the Open Transport subsystem. The `freemsg` call is not accessible to software outside the Open Transport family. Each FPI is supported by routines in a **family library.**
- **Low-level expert:** Low-level experts are software utilities that install entries in the Name Registry for specific devices. Low-level experts may reside in system firmware, PCI card firmware, or Mac OS and may run at any time. A low-level expert's task is to install enough information in each Name Registry entry to permit device control and driver matching. The information must be presented to Registry clients in a generalized form, independent of hardware configuration. Primary clients of the Registry at present are run-time device drivers and family experts (defined below).
- **Name Registry:** The Name Registry is a high-level Mac OS service that stores the names and relations of hardware and software components in the system that is currently running. In the second generation of Power Macintosh, the Name Registry is used only for I/O device and driver information, serving as a rendezvous point between low-level or hardware-specific experts and family experts. The Registry supports both name entry management and information retrieval.
- **Physical device:** A physical device is a piece of hardware that performs an I/O function and is controlled by a device driver. An example of a physical device is a video graphics card.
- **Property:** Each piece of information associated with an entry in the Name Registry is called a *property.* For example, a `driver-descriptor` property is associated in the Registry with each device that has a unique associated driver. It contains the driver description data structure described in "Native Driver Package" (page 197).
- **ROM-based driver:** ROM-based drivers are FCode drivers that are stored in an expansion ROM. They are the only kind of drivers that are usable when

the system is starting up and the file system is not yet available, as described in Chapter 5, "PCI Open Firmware Drivers." Expansion ROMs may in addition to a basic FCode driver contain a native run-time driver for Mac OS, stored as a CFM fragment; run-time drivers are described in Chapter 8, "Writing Native Drivers."

- **Scanning:** Scanning is the process of matching a device with its corresponding driver. Scanning to determine device location and driver selection is one of the topics discussed in this chapter.
- **Driver services:** The driver services are a set of services that Mac OS provides for drivers or other pieces of software that are installed and run in the operating system. For example, `QueueSecondaryInterruptHandler` is a driver services routine in Mac OS that defers interrupt processing. Application-level software does not generally have access to the driver services. For more information about the Macintosh driver services for PCI cards, see Chapter 11, "Driver Services Library."
- **Virtual device:** A virtual device is a piece of code that provides an I/O capability independently of specific hardware. An example of a virtual device is a RAM disk. A RAM disk performs disk drive functions but is actually just code that reads and writes data in the system's physical memory.

Concepts

To prepare for changes in current and future releases of Mac OS, Apple is introducing several new or modified concepts in the second generation of Power Macintosh computers. The concepts include

- separation of application and system services
- common packaging of loadable software
- the Name Registry
- families of devices
- ROM-based and disk-based drivers
- noninterrupt and interrupt-level execution
- generic and family drivers

- driver descriptions

These concepts are discussed in the next sections.

Separation of Application and Driver Services

Previous versions of Mac OS had only one kind of operating-system interface, an application programming interface (API). This meant that all Mac OS services were available to all varieties of Macintosh software. With the introduction of Power Macintosh computers with the PCI bus, Apple starts distinguishing between APIs and driver services. The distinction must be made because programming contexts are becoming increasingly specialized as Mac OS evolves.

In the native driver model, Toolbox services (for example, the `ModalDialog` function and Menu Manager calls) are not available to drivers. Drivers operate outside the user interface and the application software environment.

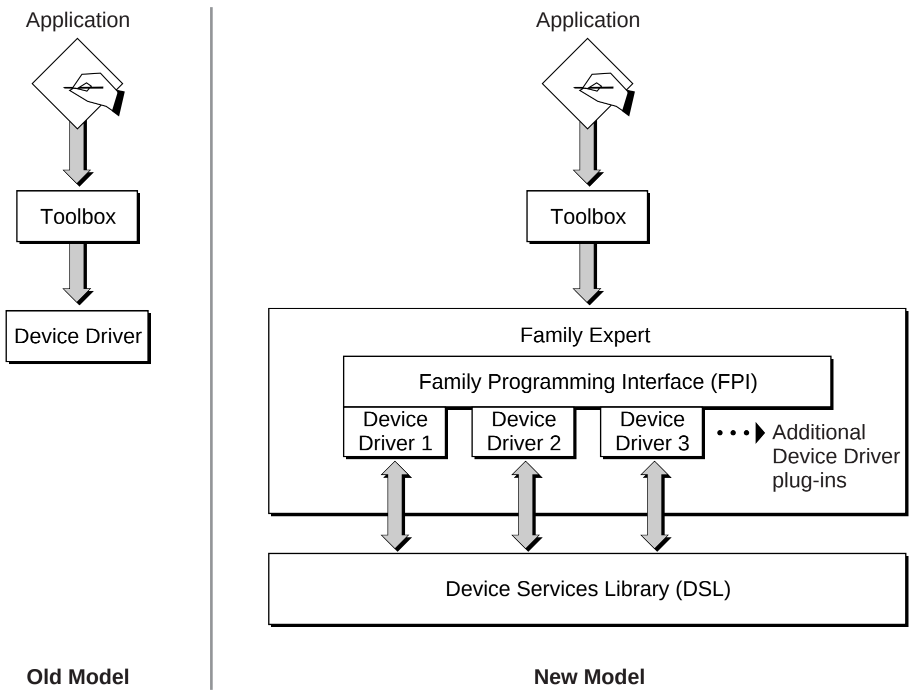
Note
The Device Manager presents an API for generic drivers, as described in "Generic and Family Drivers" (page 151). ◆

Family services required by device drivers are provided by family experts, using family libraries. These services are also not available to applications.

The separation of application and driver services in Mac OS is a big change that starts with the Power Macintosh computers with the PCI bus architecture. The difference between the old API model and the new API/driver services model is diagrammed in Figure 6-1 (page 145).

To help ensure compatibility with future releases of the Mac OS, native drivers should not use Mac OS Toolbox APIs. However, there may be instances where use of such calls is difficult to avoid, but native drivers using Toolbox calls may not work in the future. An example of using the new native driver APIs as opposed to the Toolbox APIs would be to use the Name Registry functions to store device configuration information rather than the Resource Manager. See "Driver Migration" (page 177) and "Device Configuration" (page 182) for additional information.

Figure 6-1 New system model



Common Packaging of Loadable Software

Native device drivers are created as CFM fragments. Each CFM fragment exports a driver description structure that the system uses to locate, load, and initialize the driver. Previously, device drivers were created as Mac OS resources.

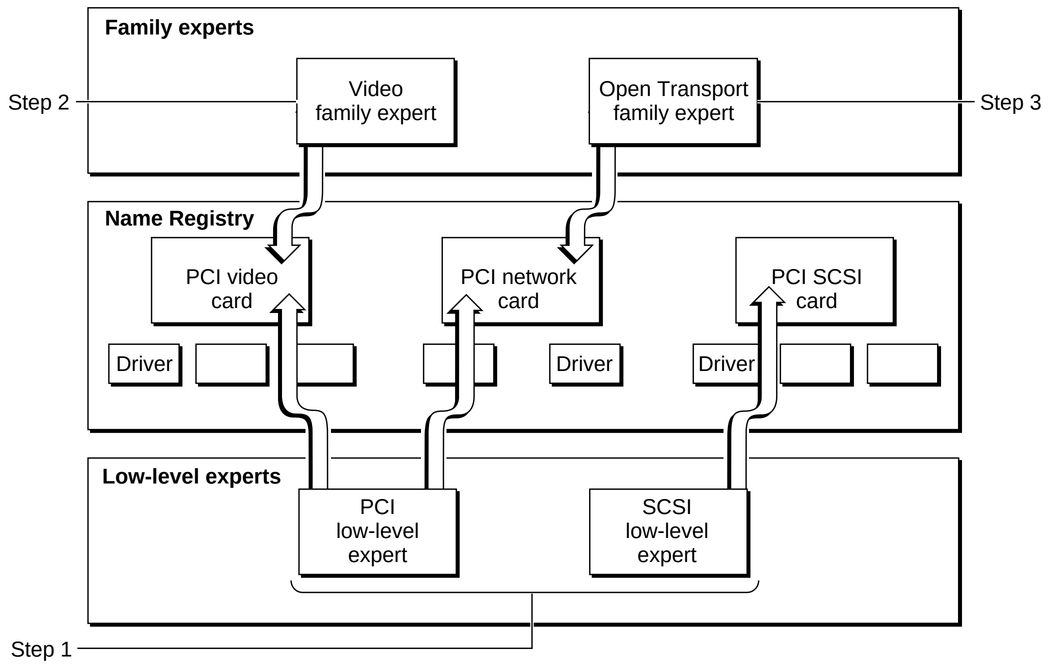
Hence native drivers are packaged differently from previous Mac OS drivers. Because they are CFM fragments, they have easy access to global data storage, and they can be written in a high-level language without assembly-language headers. Each instance of a single driver has private static data and shares code with every other instance of that driver. The CFM is responsible for maintaining the driver context (similar to the “A5 world” in previous Macintosh programming). A device driver no longer locates its private data by means of a field in its device unit table entry.

One consequence of drivers as CFM code fragments is that a single device driver no longer controls multiple devices. Normally there is an instance of the driver for each device, although only one copy of the driver's code is loaded into memory.

The Name Registry

The Mac OS **Name Registry** is a database of system information. The native I/O framework uses the Registry as a general storage and retrieval mechanism for family experts and low-level experts. Device scanning code and the Name Registry help separate system initialization and device driver initialization in a well-defined way, as illustrated in Figure 6-2. The Name Registry is more fully described in Chapter 10, "Name Registry.".

Figure 6-2 Typical role of the Name Registry



Although it does not drive the startup process, the Name Registry assists system startup by providing a structure for storing information. It does this in several ways:

- During the computer's startup process, low-level experts in the Mac OS ROM and in expansion ROMs install and update system information in the Name Registry.
- Other software in the startup process can then use the Name Registry to locate devices required to initialize the system.
- System firmware installs disk-based drivers and other system components in the Name Registry when the file system becomes available.
- Disk-based experts can then use information in the Name Registry to locate and initialize family devices.
- When device initialization driver code is called, the Name Registry provides configuration information for device drivers and family experts.

These processes are marked by steps in Figure 6-2. In Step 1, low-level experts scan the PCI bus for their device types and create name entries in the Name Registry that identify device properties and contain device drivers. In Step 2, family experts locate all name entries that match their service categories. In Step 3, family experts obtain device drivers and call the drivers' initialization routines.

To make driver design easier, the Name Registry lets all types of device drivers be written identically, whether they are located in expansion ROMs, system firmware, or elsewhere. Drivers can expect basic hardware information to be available in the Registry and are not required to locate or hard code this data.

The Name Registry supports a comprehensive driver replacement capability, described in Chapter 7, "Finding, Initializing, and Replacing Drivers." All device entries and their corresponding code drivers exist in the device portion of the Name Registry and are available for this process.

Families of Devices

Families are groups or categories of devices that provide similar or the same functionality and have the same basic software interface requirements. An example of a device family is the set of devices that provide networking services to the system. These devices are not the same—for example, Ethernet is not the same as LocalTalk—but they all run within the Open Transport family and use the Open Transport libraries to augment the driver services provided by Mac OS. A second example of a device family is the set of all display devices. The concept of device family is critical to the native driver model because it allows the needs of each device family to be met independently of the needs of other families.

Mac OS provides built-in support for device families such as the display family and the network family. Each of these device families has access to services that isolate system and application software from particular device characteristics. For example, the **Display Manager** provides a uniform programming interface—a family programming interface (FPI)—for display devices regardless of their physical form. Similarly, the Open Transport subsystem isolates the remainder of the system and applications from the particular characteristics of network devices. These FPIs are provided by family libraries in Mac OS.

The Display Manager and video drivers illustrate how a family of devices can provide and utilize family-specific services. These services are complementary to the services provided by the system software, because they are used by the family but are not duplicated by the system and are not available to other components of the system or to Macintosh applications. For a fuller discussion, see Chapter 13, "Graphics Drivers."

A family expert such as the Open Transport expert interrogates the Name Registry for devices of a certain service category, verifying only that they are of the right category. For example, a software loopback virtual device could appear in the Name Registry, the driver for which would take data from a source and return it back to the same source. To install a loop-back Name Registry entry, the loopback configuration software would call the Name Registry to create an entry and to add the driver descriptor property with its driver information containing the appropriate service category. In networking, the service category for a loopback device is `'otan'`. Installing the loopback entry would be the work of a low-level expert for loopback devices. The family expert for Open Transport would locate the loopback entry using Registry calls, and it would initialize the driver in the Open Transport subsystem using family-specific initialization mechanisms.

ROM-Based and Disk-Based Drivers

ROM-based drivers are stored in expansion ROMs. Disk-based drivers are located in the Mac OS file system, in the Extensions folder.

ROM-based drivers with the correct information in their driver description structures are installed and opened by the Macintosh firmware, acting as the driver's client. These are the only drivers available at the beginning of system startup.

Disk-based drivers are located and opened as needed. Once the file system is working, Mac OS can replace outdated ROM-based drivers with disk-based drivers. Experts that control disk-based drivers locate and initialize their drivers soon after. Drivers that are disk-based but not under expert control, and that are not needed by Mac OS during startup, remain uninitialized and unopened until a specific client requests access to the device associated with the driver.

Noninterrupt and Interrupt-Level Execution

In prior releases of Mac OS there has been no clear distinction between application-level programming and system-level programming. Restrictions about when certain system services can be used and when they cannot are not fully defined.

In native driver model, different execution levels have different restrictions. Noninterrupt (task level) execution may make use of nearly any operating-system or Mac OS ROM Toolbox service. Secondary interrupt routines and hardware interrupt handlers are allowed only a small subset of those services.

The discussion in this book uses the following definitions for execution levels:

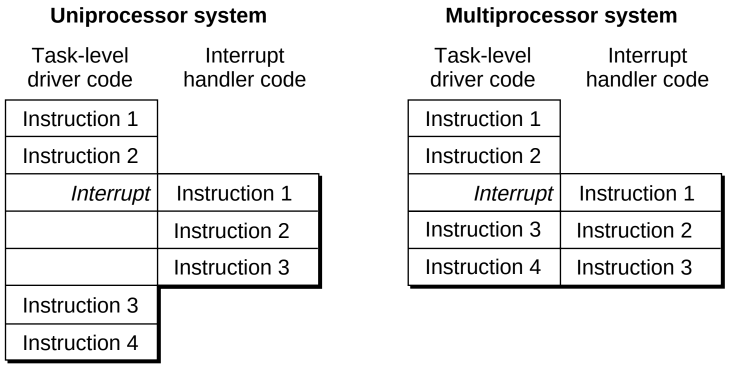
- **Task level:** the noninterrupt level on which most code, including applications, is executed.
- **Hardware interrupt level:** the execution level of concern to driver writers. Hardware interrupt-level execution happens as a direct result of a hardware interrupt request. The software executed at hardware interrupt level includes installable interrupt handlers for PCI and other devices as well as interrupt handlers supplied by Apple.
- **Secondary interrupt level:** the execution level similar to deferred task execution in previous versions of Mac OS. The secondary interrupt queue is filled with requests to execute subroutines that are posted for execution by hardware interrupt handlers. The handlers need to perform certain actions but choose to defer the execution of those actions to minimize interrupt-level execution. Unlike hardware interrupt handlers, which can be interrupted by

hardware interrupts and can nest, secondary interrupt handlers always run serially.

Symmetric Multiprocessing

In future Power Macintosh computers that feature symmetric multiprocessing (SMP), a device driver will not be able to assume that hardware or secondary interrupt level execution preempts all task level execution. In a four-processor system, for example, one processor might be running a hardware interrupt handler, another running a secondary interrupt handler, and the other two running tasks. This behavior is different from that of a uniprocessor system, where an interrupt handler normally runs to completion between two task-level instructions. The difference is illustrated in Figure 6-3.

Figure 6-3 Uniprocessing and multiprocessing execution



Symmetric multiprocessing changes some of the programming rules for driver writers. Observe these cautions:

- If you use an atomic operation to reference a particular memory location at task level (such as an atomic increment to a counter), you must also use atomic instructions when referencing that location at hardware and secondary interrupt level.
- If you disable interrupts and use secondary interrupt level following the rules in this book, you shouldn't have any problems. If you assume that no

task can be running while your interrupt handler runs, your code will break on a multiprocessor system.

- If your driver disables interrupts for its device while running at task level, an interrupt for a different device can still occur. The second interrupt may run concurrently with your task-level device driver as shown in Figure 6-3.

Disabling hardware interrupts for synchronization purposes works safely in an SMP environment. Disabling hardware interrupts on one processor guarantees that interrupts are off on every processor and that no other processor can execute code that runs with interrupts off. If another processor tries to disable interrupts, it will loop while waiting for the first processor to turn interrupts on again. This feature makes it critical that interrupts be disabled for very short periods of time.

Similarly, in an SMP environment only one processor at a time can run at secondary interrupt level. Other processors trying to run at secondary interrupt level will do no useful work until the first processor exits that level. For this reason, secondary interrupt level should be used as sparingly as possible.

Generic and Family Drivers

The Macintosh native driver model defines a new driver packaging format, described in “Driver Package,” later in this chapter. The driver package may contain a generic driver or a driver that is specific to a family of devices. **Generic drivers** have a family type of `'ndrv'` and are controlled by the Device Manager (described in *Inside Macintosh: Devices*). Family drivers have other type designations and do not act as Device Manager clients. They are not installed in the Device Manager unit table and do not export the generic driver call interface. The generic driver call interface and runtime framework are described in “Generic Driver Framework” (page 153).

Currently most drivers are generic, but this will not be true in future versions of the Mac OS. Some drivers belong to device families with special characteristics that do not fit into the generic driver model; they are drivers controlled by family experts. Examples of this type of driver are FireWire SBP-2 drivers, USB drivers, and networking device drivers for the Open Transport environment. Networking device drivers under Open Transport are STREAMS drivers that provide industry standard STREAMS/DLPI interfaces to Mac OS. Open Transport drivers are discussed in Chapter 14, “Network Drivers.” USB drivers are discussed in the documentation included in the Mac OS USB DDK.

Drivers controlled by family experts use family programming interfaces (FPIs), which are defined for each family and are not accessible to Macintosh applications.

All drivers with driver family service interfaces must export well-defined driver family service names for both family expert data and family expert functions. Clients of family drivers load the CFM-based driver and call the exported names. The CFM connects the driver client to the CFM device driver exports. Native device drivers that provide private family interfaces need not provide an additional generic driver interface to Mac OS.

As an example of a family interface, Open Transport requires a family data structure called `install_info` and a driver family service function whose name is `GetOTInstallInfo`. The `install_info` structure is used by Open Transport to create stream to STREAMS device drivers. The Open Transport family expert calls the device driver family expert `GetOTInstallInfo` function as part of the installation process for native drivers of the `'otan'` service category. See Chapter 14, “Network Drivers,” for more details on Open Transport driver requirements.

Other family drivers are described in Chapter 13, “Graphics Drivers,” and Chapter 15, “SCSI Drivers.”.

Note
Device drivers need to provide only one family interface. If a device driver chooses to provide more than one service category programming interface, however, it must conform to the standards of each interface. ◆

Driver Descriptions

Drivers are CFM code fragments and must export driver description structures to characterize their functionality and origin. The structures must be exported through the CFM’s symbol mechanism, using the symbol name `TheDriverDescription`. The complete structure is defined in Chapter 9, “Driver Loader Library.” It is based on the `driver-descriptor` property associated with device entries in the Name Registry, described in Chapter 10, “Name Registry.”

Note
Open Firmware uses the `driver-descriptor` property and the Mac OS uses `TheDriverDescription` symbol to refer to the same structure. ◆

The `DriverDescription` structure is used by scanning code to

- match Registry entries to drivers
- identify device entries by service type or family
- provide driver version information
- provide driver initialization information
- allow replacement of ROM-based drivers with newer disk-based drivers

By providing a common structure to describe drivers, the system is able to regularize the mechanisms used to locate, load, initialize, and replace them. Details of how this works are given in Chapter 7, "Finding, Initializing, and Replacing Drivers."

Mac OS treats any CFM code fragment that exports a driver description structure as a native driver.

Generic Driver Framework

This section describes the system software framework in the second generation of Power Macintosh for generic run-time drivers—that is, drivers of family type `'ndrv'`.

Device Manager

The traditional Mac OS **Device Manager** controls the exchange of information between applications and hardware devices by providing a common programming interface for applications and other software to use when communicating with generic device drivers. Normally, applications don't communicate directly with generic drivers; instead, they call Device Manager functions or call the functions of another manager that calls the Device Manager.

In the second generation of Power Macintosh, two significant additions have been made to the Device Manager. First, drivers can now process more than one request simultaneously. Such drivers are called **concurrent drivers.** Second, a new entry point has been added, similar to `IODone`. It is called `IOCommandIsComplete`. Details on concurrent drivers and their use of `IOCommandIsComplete` are given in "Completing an I/O Request" (page 191).

Driver Package

The native driver model defines a new driver packaging format. This package may contain generic drivers or family drivers, as explained in "Generic and Family Drivers," earlier in this chapter.

The native driver package is a CFM code fragment that may reside in the Macintosh ROM, in an expansion ROM, or in the data fork of a Preferred Execution Format (PEF)file. File-based generic and family driver fragments do not need a resource fork, have a file type of `'ndrv'`, and have an unspecified file creator. ROM-based PCI drivers may be replaced by disk-based versions of the driver located in the Extensions folder. PEF and the CFM are described in *Inside Macintosh: PowerPC System Software.*

A native driver package must define and export at least one data symbol through the CFM's export mechanism. This symbol must be named `TheDriverDescription`; it is a data structure that describes the drivers type, functionality, and characteristics. This data structure is described in "Driver Description Structure" (page 198).

Depending on the type of driver, additional symbols may need to be exported. The generic `'ndrv'` driver type requires that the CFM package export a single code entry point called `DoDriverIO`, which passes all driver I/O requests. `DoDriverIO` must respond to the k`OpenCommand`, k`CloseCommand`, k`ReadCommand`, k`WriteCommand`, k`ControlCommand`, k`StatusCommand`, k`KillIOCommand`, k`InitializeCommand`, and k`FinalizeCommand` commands. Other driver types for device families export and import symbols and entry points defined by the device family or device expert.

Driver Services Library

The native driver framework includes a **Driver Services Library (DSL)** that supplies the driver services required by most generic drivers. Driver services are described in "Separation of Application and Driver Services" (page 144). The driver loader links the DSL automatically to each generic driver at load time. Mac OS may provide additional services to drivers in certain families or categories.

The types of services represented in the Driver Services Library include

- request processing services
- memory management services

- interrupt management services
- secondary interrupt handlers
- atomic operation services
- timing services
- operating system utilities

The calls supplied by the DSL and the family support libraries constitute the complete set of services provided to device drivers. The calls in the DSL are the only driver interfaces guaranteed to be maintained in subsequent releases of Mac OS. Calls to services outside of the DSL and the family support libraries (for example, calls to Toolbox traps, low-memory globals, and similar vectors) will result in driver failure.

Converting Previous Mac OS Drivers

This section introduces the issues involved in the conversion of 68K Mac OS drivers to native drivers.

Restricted Access to Services

As mentioned in "Separation of Application and Driver Services" (page 144), the native driver model distinguishes between APIs and driver services. Services such as modal dialog displays or Menu Manager calls are not available to native drivers; instead, drivers will use only the interfaces provided by the Driver Services Library and driver family expert interfaces for the device. Those parts of a 68K driver that require services provided by the Macintosh Toolbox must be removed from the driver and placed in a standalone application or control panel.

In addition to restricting the types of Toolbox calls drivers are able to make, there are changes to existing mechanisms that will allow drivers written for the second generation of Power Macintosh to be used unchanged in the subsequent releases of Mac OS.

The section "Driver Migration" (page 177) documents the programming interface changes between previous Mac OS driver calls and the native driver services provided for PCI drivers. It also lists the replacement calls for existing mechanisms.

Error Returns

As is always the case with programming interfaces, native driver code should check the error returns from calls to system services. The new driver model includes the following 32-bit error return type:

```
typedef long OSStatus;
```

The lower 16 bits of `OSStatus` are equivalent to the existing `OSErr` type, described in *Inside Macintosh: Overview.* In current versions of Mac OS, the upper 16 bits contain the sign extension of the lower 16 bits. At present there are just two possible values for the upper 16 bits, all 1s or all 0s; other values are reserved for future use by Apple.

Ensuring Future Compatibility

You should take steps to ensure that your driver is compatible with current and future releases of Mac OS. Two steps you can take are as follows:

- Substantial changes in task execution and interrupt handling affect native drivers. The tasking model and interrupt handling mechanisms will be increasingly hidden behind the Driver Services Library, the Driver Loader Library, and the Name Registry. Drivers that do not use the native libraries provided with the current release of Mac OS may not work with subsequent releases.
- In the current Mac OS environment there is one address space, which all applications, Toolbox services, and drivers share. In future versions of Mac OS there may be many address spaces, and applications and their associated data may exist outside the address space in which the kernel, driver services, and drivers exist. It is not possible to verify correct address space usage using the current version of Mac OS, but strict adherence to the rules outlined below will guarantee compatibility with future releases.

Task execution and interrupt handling are discussed in detail in various sections of Chapters 8 through 11. Toolbox services that are not available to native drivers are listed in “Driver Services That Have No Replacement” (page 177). Addressing problems are discussed next.

Copying Data

To allow compatible driver development on the current version of Mac OS, future releases of Mac OS will give drivers that are managed by the Device Manager a restricted set of facilities for mapping address spaces and copying data from one space to another. Device families, such as video displays, will have additional family-specific facilities to address their data transfer needs. Hence, drivers that exchange data with applications via the Device Manager must do so via `PBRead` and `PBWrite` calls. Depending on the size of the data buffer, the Device Manager will copy or map the `IOParamBlockRec` data structure for these calls and will copy or map the associated `ioBuffer` up to `ioReqCount` bytes.

`PBOpen`, `PBClose`, `PBControl`, `PBStatus`, and `PBKillIO` calls will keep the `IOParamBlockRec` and `CntrlParam` data structures accessible; however, no referenced data will be copied or mapped. This means that the `csParam` fields of the `CntrlParam` block must not contain buffer pointers to additional data, and the `ioBuffer` field will be ignored for Device Manager calls (such as `PBOpen` and `PBClose`) for which it is not a documented input parameter. The Device Manager will not copy or map data pointed to by either of these fields.

In the past, applications and device drivers have extended the size of the `IOParamBlockRec` and `CntrlParam` structures to tag additional information into a device driver request. This was possible because applications and device drivers shared a single address space. In future releases of Mac OS, the Device Manager will copy only the `IOParamBlockRec` and `CntrlParam` structures as defined in *Inside Macintosh: Devices.*

Power Management

The Mac OS Power Manager API is still being defined and is likely to change in future releases of Mac OS. You are encouraged to make use of the power management facilities in family experts and device specific managers instead. If you must use the Power Manager, be careful to use only its published API.

Summary

The I/O architecture defined in this chapter sets a durable standard for writing Mac OS device drivers. This standard is supportable in future releases of Mac OS, and device drivers that conform to it may work unmodified and efficiently with those releases. Successful execution of this strategy, which allows native device drivers to work portably and effectively across future Mac OS releases, depends upon the successful adoption of the guidelines summarized in this section.

Use the System Programming Interfaces

The use of driver services is essential to a driver's portability to future Mac OS releases. These are the programming interfaces for device drivers that are guaranteed to be common across Mac OS system versions. They consist of

- The Name Registry library `NameRegistryLib`
- The Driver Services library `DriverServicesLib`
- A service library specific to each high-level device family

When writing a device driver, never use Toolbox API calls. Doing so will prevent your device driver from being compatible with future Mac OS releases. Instead, use the functionality provided by the corresponding drivers services APIs, for example when working with USB devices, use the USB Services Library (USL) and USB Manager. The driver services calls let you deal more naturally with device driver issues that the Toolbox API does, because the Toolbox is intended for applications.

You can ensure compliance with the foregoing rule by not letting your driver link with application libraries such as `InterfaceLib`, `MathLib`, `StdCLib`, and so on. Any necessary Toolbox functionality, such as driving a graphical user interface, should be accomplished by separate application-level software on behalf of the device driver.

Use the Name Registry

The Name Registry provides a unified way of identifying or obtaining information about many system resources, not just about devices. When writing a device driver, never acquire information from low memory or through Toolbox API calls because doing so may prevent your driver from being compatible with future Mac OS releases. Instead, use the Name Registry to acquire the information. During driver initialization, family experts facilitate this process by passing each driver a `RegEntryID` representing its physical device. By using the `RegEntryID` and the Name Registry a device driver can find all the information it is likely to need.

For further information about the Name Registry, see Chapter 10, “Name Registry.”

C H A P T E R 7

Finding, Initializing, and Replacing Drivers

The native driver framework in PCI-based Power Macintosh computers tolerates a wide range of variations in system configuration. Although drivers and expansion cards may be designed and updated independently, the system autoconfiguration firmware offers several techniques for making them work together. This chapter discusses what PCI driver and card designers can do to improve the compatibility of their products.

Device Properties

A PCI device is required to provide a set of properties in its PCI configuration space. It may optionally supply FCode and run-time driver code in its expansion ROM. PCI devices without FCode and run-time driver code in ROM cannot be used during system startup.

The required device properties in PCI configuration space are

- `vendor-ID`
- `device-ID`
- `class-code`
- `revision-number`

For PCI boot devices there must be an additional property:

```
driver,AAPL,MacOS,PowerPC
```

This property contains a pointer to the boot driver's image in the PCI card's expansion ROM. It is used in conjunction with the `fcode-rom-offset` property.

The Open Firmware FCode in a PCI device's expansion ROM must provide and install a `driver` property, as shown above, to have its driver appear in the Name Registry and be useful to the system during startup. It must also add its expansion ROM's base register to the `reg` property, so that system firmware can allocate address space when installing the driver.

To facilitate driver matching for devices with disk-based drivers, the FCode should provide a unique `name` property that conforms to the PCI specification. For further information, see Chapter 5, "PCI Open Firmware Drivers."

PCI Boot Sequence

To better explain the concepts and mechanisms for finding, initializing, and replacing PCI drivers, below is a short description of the PCI boot sequence. The boot sequence applies to machines built prior to the introduction of the NewWorld architecture in the iMac and later Power Macintosh G3 computers. The boot sequence is similar for all Macintosh computers, however the steps are not identical. For information about the steps in the NewWorld boot sequence, see Chapter 3, "Introduction to the NewWorld Architecture," and "Startup Sequence in the NewWorld Architecture" (page 92).

1. Hardware is reset.
2. Open Firmware creates the device tree. This device tree is composed of all the devices found by the Open Firmware code, including all properties associated with those devices.
3. The Name Registry device tree is created by copying the Macintosh-relevant nodes and properties from the Open Firmware device tree.
4. The Code Fragment Manager and the interrupt tree are initialized.
5. Device properties that are persistent across system startups and are located in NVRAM are restored to their proper location in the Name Registry device tree.
6. The Name Registry device tree is searched for PCI expansion ROM device drivers associated with device nodes.
7. PCI expansion ROM device drivers required for booting are loaded and initialized.
8. If a PCI ROM device driver is marked as `kDriverIsLoadedUponDiscovery`, the driver is installed in the Device Manager unit table.
9. If a PCI ROM device driver is marked as `kDriverIsOpenedUponLoad`, the driver is initialized and opened, and the `driver-ref` property is created for the driver's device node.
10. The Display Manager is initialized.
11. The ATA Manager is initialized.
12. The SCSI Manager is initialized.

13. The File Manager and Resource Manager are initialized.
14. Device properties that are persistent across system startups and located in the Preferences folder in the System Folder are restored to their proper location in the Name Registry device tree.

Device drivers under family expert control are processed next. The following steps load disk-based experts and disk-based drivers:

1. Scan the Extensions folder for drivers (file type `'ndrv'`), updating the Registry with new versions of drivers as appropriate. For each driver added or updated in the tree, a driver description property is added or updated as well.
2. For each driver that is replaced, and already open, use the driver replacement mechanism.
3. Run `'INIT'` resources for virtual devices. Virtual devices are discussed in "Real and Virtual Devices" (page 285).
4. Scan the Extensions folder for experts; load, initialize, and run each expert.
5. Run experts to scan the registry, using the driver description property associated with each node to determine which families the devices belong to.
6. Load and initialize appropriate devices based on family characteristics.

At that point all devices in use by the system and family subsystems are initialized. Uninitialized and unopened devices or services that may be used by client applications are located, initialized, and opened at the time that a client makes a request for the devices or services.

Note
Native device drivers are ordered to switch from low-power to high-power mode when their devices are opened. ◆

Matching Drivers With Devices

Mac OS matches drivers to devices by using the following algorithm:

- When a device node has a driver in ROM, no driver matching is required. Mac OS uses the driver name and compares the version numbers of ROM-based and disk-based drivers to select the newest version of the driver.
- When a device node has a `name` property that was supplied by the FCode in a device's expansion ROM, Mac OS checks the name property against all disk-based drivers and finds the first matching driver with the latest version number. If there is no match against the `name` property, then Mac OS attempts a match against each name string in the device's `compatible` property. The comparison is always against the `nameInfoStr` parameter in the driver description structure for each disk-based driver. The first match is used. If no match is found against `name` or `compatible` strings, the device is not usable.
- When a device node has no FCode, Mac OS tries to match the device with a driver based on the generated name `pci`*xxxx,yyyy* where *xxxx* is the vendor ID and *yyyy* is the device ID. Both these ID values must be hexadecimal numbers, without leading 0s, that use lower case for the letters A through F and are rendered as ASCII characters. If a match is found, but the first initialization call to the driver fails, then the code that is attempting to use the driver must call the Driver Loader Library's best match routine (again) to find the next-best driver.

Note
Each device node should have just one `compatible` property, containing one or more C-formatted name strings as its value. The strings must be packed in sequence with no unused bytes between them and should be arranged with the more compatible names first. ◆

The DLL routines `GetDriverForDevice`, `InstallDriverForDevice`, and `FindDriversForDevice` use the following algorithm to match or install the "best" driver for a device:

1. Find all candidate drivers for the device. A driver is a candidate if its `nameInfoStr` value matches either the device's name or one of the names found in the device's `compatible` property.
2. Sort this list based on whether the driver matched using the device's name or a compatible name. Those matched with the device name are put at the head of the list. Ties are broken using the driver's version number (See "HigherDriverVersion" (page 269).) Pseudo code for file-based driver sorting is shown in Listing 7-1. The code returns 0 if two drivers are equally

compatible, a negative number if `driver1` is less compatible than `driver2`, and a positive number if `driver1` is more compatible than `driver2`.

3. If not installing the driver, return the driver at the head of the candidate list and discard any remaining candidates.

If you still have candidates with which to attempt an installation, do the following:

1. Load and install the driver located at the head of the list.
2. The driver should probe the device, using DSL services, to verify the match. If the driver did not successfully initialize itself, discard it and return to step 1.
3. Discard any remaining candidates.

The routines that use this algorithm are described in detail in the sections that start with “Loading and Unloading” (page 248).

Listing 7-1 File-based driver sorting

```
SInt16 CandidateCompareRoutine
            (FileBasedDriverInfoPtr                 Driver1,
             FileBasedDriverInfoPtr                 Driver2,
             StringPtr                              CompatibleNames,
             ItemCount                              nCompatibleNames)

{
    SInt16          matchResults = 0;

    if ( Driver1 and Driver2 matched using same property (name or compatible))
    {

        if ( both drivers matched using compatible property )
        {

            if ( drivers not matched with identical compatible name )
            {

                // Which compatible name (by number) did driver1/driver2 match?
                Driver1CompatibleName = WhichCompatibleName(Driver1,...);
                Driver2CompatibleName = WhichCompatibleName(Driver2,...);
                if ( Driver1CompatibleName != Driver2CompatibleName )
                {
                    if ( Driver1CompatibleName < Driver2CompatibleName )
                        return  1; // driver1 is "more compatible"
                    else
                        return -1; // driver2 is "more compatible"
                }
            }
        }

    // Break tie with version numbers, if possible.
    matchResults = HigherDriverVersion              (&Driver1 ->
    info.theType.version, &Driver2 -> info.theType.version);

    // Same version number too?
    if ( matchResults == 0 )

        {
        // Final tie breaker is their filenames
        // (Reverse the compare with RelString)
        matchResults = RelString            (Driver2 -> info.theSpec.name,
                                    Driver1 -> info.theSpec.name, true, true );
        }

    return matchResults;
    }
    // Matched using different property
    if ( Driver1 matched using compatible property )
    return -1;                                  // driver 2 is higher
    return 1;                                   // else driver 1 is higher
}
```

Driver Initialization and Resource Verification

After finding a match between a hardware device and its driver, the driver initialization code must check to make sure that all needed resources are available. This section describes a typical algorithm for resource verification. Driver initialization code should perform this algorithm for two reasons:

- The driver may not have all the address resources it requires. This event is unlikely, but the driver should check.
- If the PCI card expansion ROM doesn't contain FCode, the driver may need to perform a diagnostic to make sure the card it has been matched with is actually the card it is designed to control. This problem is discussed in "Open Firmware FCode Options" (page 87).

IMPORTANT

The driver must enable the card's address space for a PCI device to be usable. ▲

The following is a typical resource verification and card address enabling procedure:

1. Check for existence of an `assigned-addresses` property for the device. If no `assigned-addresses` property exists, exit the driver initialization routine with an error message (address resources not available). The `assigned-addresses` property is discussed in "Standard Properties" (page 322). If an `assigned-addresses` property exists, go to step 2.
2. Check the `assigned-addresses` property for the existence of the base registers required for full operation of the driver. Do this by looking at the last byte of the first long word of each `assigned-addresses` entry that is required. A typical assigned-addresses entry looks like this:

   ```
   82006810 00000000 80000000 000000000 00008000
   81006814 00000000 00000400 000000000 00000100
   ```

   If the required base registers are not present, exit the driver initialization routine with an error message (address resources not available). If the required base registers are present, continue.
3. Note where in the `assigned-addresses` property the entries for the required base registers are located. The first entry is 0, the next is 1, and so on. That same order will be preserved in the `AAPL,address` property, which is an array of 32-bit values corresponding to the logical address for your base register's physical address. For more information about the `AAPL,address` property, see "Fast I/O Space Cycle Generation" (page 454). A typical `AAPL,address` property looks like this:

   ```
   80000000 F2000400
   ```

   If the driver uses Expansion Bus Manager routines (such as

`ExpMgrIOReadByte`) it must pass the physical address for the I/O base register, which it gets from the `assigned-addresses` property. The Expansion Bus Manager does byte swapping and EIEIO synchronization for the driver, but it's node-based and it's slow. The `AAPL,address` version just uses a pointer, so it's as fast as accessing memory space.

4. If the driver can be confused with another driver—if, for example, the card doesn't have FCode and another vendor uses the same PCI ASIC on a different card—the driver must perform a diagnostic routine on the card to make sure that it has been matched correctly. The `DeviceProbe` function, described below, helps a driver determine if a device is present at an address. If the diagnostic routine fails, the driver must exit its initialization routine with an error message (not my card). If the driver verifies that the card is correct, continue.
5. The driver must read or write to the device's configuration command register to enable its PCI spaces. Listing 7-2 presents typical code for doing this. It uses the `ExpMgrConfigReadWord` routine (page 462).

Listing 7-2 Enabling PCI spaces

```
ExpMgrConfigReadWord (yourNode, 4, &yourvalue);
yourvalue = yourvalue | yourEnables;          /* if I/O space, bit 0;
                                                 if memory space, bit 1 */
ExpMgrConfigWriteWord (yourNode, 4, yourvalue);
```

Listing 7-3 shows a routine that extracts a device's logical address by using its `assigned-addresses` and `AAPL,address` properties. It accepts as input the offsets into PCI configuration space that match the device's space request. For example, an Ethernet card may want two address spaces, I/O and memory. The card is designed so that offset 0x10 in configuration space corresponds to the I/O space and 0x14 corresponds to the memory space.

Listing 7-3 Getting a device's logical address

```
// The following values are valid for offsetValues (:
//
//      #define kPCIConfigBase10Offset          0x10
//      #define kPCIConfigBase14Offset          0x14
//      #define kPCIConfigBase18Offset          0x18
//      #define kPCIConfigBase1COffset          0x1C
//      #define kPCIConfigBase20Offset          0x20
//      #define kPCIConfigBase24Offset          0x24
//      #define kPCIConfigBaseROM30Offset       0x30

// Input:
//      theID - the NameRegistry ID for a PCI card
//      baseRegAddress - no input value
//      offsetValue -   config base offset, determines which address space
//                      logical address is returned
//      spaceAllocated - no input value

// Output:
//      if err = kOTNoError, *baseRegAddress - contains the logical address of a PCI
//      address space, also spaceAllocated is a byte count for the amount of space
//      that was allocated
//      returns various errors
//
//---------------------------------------------------------------------------------

OSStatus GetPCICardBaseAddress(RegEntryID *theID, UInt32 *baseRegAddress,
                                     UInt8 offsetValue, UInt32 *spaceAllocated)
{
OSStatus                osStatus;
PCIAssignedAddress      *assignedArray;
RegPropertyValueSize    propertySize;
UInt32                  numberOfElements, *virtualArray;
Boolean                 foundMatch;
UInt16                  index;

*baseRegAddress = NULL;         // default value
foundMatch = kFalse;

osStatus = GetAProperty(theID, kPCIAssignedAddressProperty,(void **)&assignedArray,
            &propertySize);

if ((osStatus == kOTNoError) && propertySize)
   {
   numberOfElements = propertySize/sizeof(PCIAssignedAddress);
```

```
    osStatus = GetAProperty(theID, kAAPLDeviceLogicalAddress,(void **)&virtualArray,
                            &propertySize);

    if ((osStatus == kOTNoError) && propertySize)
        {
      // search through the assigned addresses property looking for base register

        for (index = 0; (index != numberOfElements) && !foundMatch; ++index)
            {
            if (assignedArray[index].registerNumber == offsetValue)

                {
                *spaceAllocated = assignedArray[index].size.lo;
                *baseRegAddress = virtualArray[index];
                foundMatch = kTrue;
                }
            }
        DisposeProperty((void **)&virtualArray);
        }
    else
        osStatus = kENXIOErr;

    DisposeProperty((void **)&assignedArray);
    }
else
    osStatus = kENXIOErr;

return osStatus;
}
```

DeviceProbe

`DeviceProbe` is used to determine if a hardware device is present at the indicated address.

```
OSStatus DeviceProbe (
                     void *theSrc,
                     void *theDest,
                     UInt32 AccessType);
```

`theSrc`	The address of the device to be accessed.
`theDest`	The destination of the contents of `theSrc`.
`AccessType`	How `theSrc` is to be accessed: `k8BitAccess, k16BitAccess, or k32BitAccess`.

DESCRIPTION

`DeviceProbe` accesses the indicated address and stores the contents at `theDest` using `AccessType` to determine whether it should be an 8-bit, 16-bit or 32-bit access. Upon success it returns `noErr`. If the device is not present, that is, if a bus error or a machine check is generated, it returns `noHardwareErr`.

If a PCI card contains no FCode, and therefore is assigned a generic name of the form `pci`*xxxx,yyyy,* it is important for a driver to provide diagnostic code in its `Initialize` routine. When a driver is matched with a card that has a generic name property, it may be the wrong driver. In that case, diagnostic code probing for a unique characteristic of the card not only may fail a data compare operation but may also cause an unrecoverable machine check exception. `DeviceProbe` allows a driver to explore its hardware in a recoverable manner. It provides a safe read operation, which can gracefully recover from a machine check and return an error to the caller. If `DeviceProbe` fails, the driver should return an error from its `Initialize` command. This return may cause the DLL to continue its driver-to-device matching process until a suitable driver is found.

RESULT CODES

`noErr`	0	Device present
`noHardwareErr`	–200	Device not present

Opening Devices

There is a clear distinction between device initialization and device opening. A device opening action is a connection-oriented response to client requests. Device drivers should expect to run with multiple `Open` and `Close` commands. This means that each driver is responsible for counting open requests from clients, and must not close itself until all clients have issued close requests. Initialization can occur independently of client requests—for example at startup time, or (in the case of PCMCIA devices) when a device is hot-swapped into or out of the system.

Initialization of native driver–controlled devices is handled in phases as described in the previous section. It is necessary to make a distinction here between PCI drivers and 68K drivers because the 68K driver initialization path has not changed.

The first phase of native driver initialization consists of searching the device portion of the Name Registry for boot devices. Boot device nodes should be flagged as `kDriverIsLoadedUponDiscovery` and `kDriverIsOpenedUponLoad` in the `driver-descriptor` property associated with the device node. The contents of the `driver-descriptor` property is a `TheDriverDescription` structure. Boot devices are loaded, initialized, and opened by the system. Their drivers, which must be in ROM, should be passive code controlled by the system starting up.

The second phase of startup comes after the Mac OS file system is available. In this second phase the Extensions folder is scanned for family experts, which are run as they are located. Their job is to locate and initialize all devices of their particular service category in the Name Registry. The family experts are initialized and run before their service category devices are initialized because the family expert extends the system facilities to provide services to their service category devices. For example, the Display Manager extends the system to provide VBL capabilities to `'disp'` service category drivers. In the past, VBL services have been provided by the Slot Manager; but with native drivers, family-specific services such as VBL services move from being a part of bus software to being a part of family software.

A family expert, whether ROM based or disk based, scans the Name Registry for devices of a particular service category. Each device entered in the Registry with the specified service category is initialized and installed in the system in a family-specific way.

Note that startup steps 10 and 11 listed on (page 163) initialize the Display Manager and the SCSI Manager. The Display Manager and SCSI Manager are both family experts. These are ROM-based experts that look for service category `'disp'` (`'ndrv'` for current display devices) and service category `'blok'` respectively. The SCSI Manager loads and activates PCI SIMs in the way described in *Inside Macintosh: Devices* and in "SIMs for Current Versions of Mac OS" (page 561). The Display Manager loads, initializes, and opens display devices. Disk-based experts perform exactly the same task as ROM-based experts, but disk-based experts are run after the file system is available. For more information about the Display Manager, see *Display Device Driver Guide*, listed in "Apple Publications" (page 26).

Driver Replacement

Suppose you are shipping your PCI card and have discovered an obscure bug in your expansion ROM driver. This section describes the mechanism that Apple supplies to allow you to update your ROM-based driver with a newer disk-based version.

As described earlier in this chapter, the Name Registry is populated with device nodes that have `driver,AAPL,MacOS,PowerPC` properties and `driver-descriptor` properties. These properties are loaded from the expansion ROM and configuration space, installed by the Open Firmware code, and pruned by the Expansion Manager.

After the Registry is populated with device nodes, the Macintosh startup sequence initializes the devices. For every device node in the Registry there are two questions that require answers before the system can complete a client request to use the device. The client may be the system itself or an application. The questions are

- Is there a driver for this node?
- Where is the most current version of the driver for this node?

If there is a driver in ROM for a device, the `driver,AAPL,MacOS,PowerPC` property is available in the Name Registry whenever a client request is made to use that device. However, after the operating system is running and the file system is available, the ROM driver may not be the driver of choice. In this case, the ROM-based driver is replaced with a newer version of the driver on disk by the following means.

In the system startup sequence, as soon as the file system is available Mac OS searches the Extensions folder and matches drivers in that folder with device nodes in the Name Registry. For a discussion of driver matching, see "Matching Drivers With Devices" (page 164). The `driverInfoStr` and `version` fields of the `DriverType` fields of the two driver description structures are compared, and the newer version of the driver is installed in the tree. When the driver is updated, the `driver-descriptor` property and all other properties associated with the node whose names begin with `Driver` are updated in the Name Registry.

If the driver associated with a node is open (that is, if it was used in the system startup sequence) and if the driver is to be replaced, the system must first close the open driver, using the `driver-ref` property in the Name Registry to locate it. The system must then update the Registry and reinstall and reopen the driver. If the close or finalize action fails, the driver will not be replaced.

The native driver model does not provide automatic replacement of 68K drivers. It is possible to replace a 68K driver, but it requires some work on your part. If you want to replace a 68K driver with a native driver dynamically, you must close the open 68K driver, extract its state information, and load and install the native driver using the Driver Loader Library. The native driver must occupy the same DCE slot as the 68K driver and use the same reference number. After being opened, it must start running with the state information that was extracted from the 68K driver.

Applications and other software can use the `ReplaceDriverWithFragment` function to replace one driver with another and `RenameDriver` to change a driver's name. These routines are described next.

ReplaceDriverWithFragment

`ReplaceDriverWithFragment` replaces a driver that is already installed with a new driver contained in a CFM fragment. It sends replace and superseded calls to the drivers, as described in "Replace and Superseded Routines" (page 218).

```
OSErr sReplaceDriverWithFragment(
                    DriverRefNum theRefNum,
                    CFragConnectionID fragmentConnID);
```

`theRefNum` Reference number of the driver to be replaced.

`fragmentConnID`
CFM connection ID for the new driver fragment.

DESCRIPTION

Given the unit table reference number of an installed driver in `theRefNum`, `ReplaceDriverWithFragment` replaces that driver with a new driver contained in a CFM fragment identified by `fragmentConnID`. It sends replace and superseded calls to both drivers, as described in “Replace and Superseded Routines” (page 218).

Note
The CFM `fragmentConnID` variable should be freed when the driver is unloaded. ◆

RESULT CODES

`noErr`	0	No error

All CFM errors (See *Inside Macintosh: PowerPC System Software*)

RenameDriver

`RenameDriver` changes the name of a driver.

```
OSErr RenameDriver(
                    DriverRefNum refNum,
                    StringPtr newDriverName);
```

`refNum`	Reference number of the driver to be renamed.
`newDriverName`	Pointer to the driver’s new name in a Pascal string.

DESCRIPTION

Given the unit table reference number of an installed driver in `refNum`, `RenameDriver` changes the driver’s name to the contents of a string pointed to by `newDriverName`.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number

Driver Migration

Driver migration is the process of converting current 68K drivers to native driver equivalents.

Resources of type `'DRVR'` in Mac OS are used to solve a broad variety of problems. Some `'DRVR'` resources drive devices as part of the I/O subsytem. For example, SCSI disk device drivers use the SCSI Manager's I/O interface to access disks on the SCSI bus. These I/O manager–based resources need not migrate to the new services and run-time model. However, the `'DRVR'` resources that control physical devices attached to a PCI bus must operate in a new, more restrictive environment.

This section covers changes to existing driver mechanisms, as well as the replacement calls. Please note that these are guidelines; for exact calling sequences and parameter descriptions you must refer to the chapters that cover each of the new routines.

Driver Services That Have No Replacement

The services described in this section are limited or not available for native driver use.

Device Manager

Native drivers are no longer part of the Toolbox environment. The implication of this change is that while 68K drivers can make `PBOpen`, `PBClose`, and `PBControl` calls, these services are not available to drivers in the native device driver environment. Drivers can make calls to other drivers through the Device Manager. Subsystems designed to communicate in this way must be reimplemented.

In the AppleTalk implementation prior to the introduction of Open Transport, the AppleTalk protocol modules are layered on top of networking device drivers using the Device Manager as the interface mechanism between these cooperating pieces of software. The native AppleTalk implementation uses UNIX®-standard STREAMs communication mechanisms to stack protocol modules on top of drivers. Network drivers are written to conform with the native device driver model and operate within the Open Transport family of devices. For further information, see Chapter 14, "Network Drivers."

Exception Manager

Native device drivers must not make calls to the Exception Manager. In the past, drivers made use of the microprocessor's bus error mechanism to probe for hardware. Drivers should instead use the Name Registry to find all devices and their properties.

Gestalt Manager

Gestalt calls are available only to applications, not to drivers. Drivers may provide `driverGestalt` services via the `Status` selector to `DoDriverIO` or may alternatively present device information through the Name Registry. The Name Registry is a unifying mechanism and is the preferred means for representing system information. See "Driver Gestalt" (page 221) for additional information about the driver Gestalt services.

Mixed Mode Manager

Native device drivers must be written entirely in native PowerPC code. Calls to the Mixed Mode Manager are not allowed. Future releases of Mac OS may not provide emulation facilities for device drivers.

Notification Manager

The Notification Manager is not currently available to native drivers, but will be available in future versions of Mac OS.

Power Manager

In general, native driver writers should exercise caution using the Macintosh Power Manager, because doing so may limit the driver's compatibility with future releases of Mac OS. In some cases, native experts provide power management facilities for client drivers, in which case native drivers should support such expert functionality.

Resource Manager

Resource Manager calls are not permitted, not even in the driver initialization routine. Instead, drivers use the Name Registry to manage initialization and configuration. The CFM provides dynamic loading of code fragments. See the discussion in Chapter 9, "Driver Loader Library."

Segment Loader

No Segment Loader calls are allowed. The Segment Loader is replaced by the Code Fragment Manager, which provides a mechanism for dynamically loading and unloading code fragments.

Shutdown Manager

Shutdown queue routines are no longer needed. The driver's CFM termination routine is called before shutdown.

Slot Manager

The Name Registry replaces the Slot Manager in most cases. For special bus-specific I/O requests, see Chapter 12, "Expansion Bus Manager."

Vertical Retrace Manager

Vertical blanking (VBL) facilities are intended to provide support to graphics and video display devices. This functionality is provided to video devices by the video display expert that is responsible for the display family. Devices outside the display family may not make VBL calls. Timing services are provided to all devices.

Native Driver Services

This section describes services that the Mac OS provides for native drivers.

Registry Services

Chapter 10, "Name Registry," introduces the concept of the Name Registry. The Registry interface provides new mechanisms for drivers to expose information. Any data that might be of use to a configuration or debugging utility may be installed in the Registry and is directly available to the configuration application through the Registry programming interface.

▲ **WARNING**
Only a small set of Registry services are available at hardware or secondary interrupt level. The set of services available at nontask level are gets and sets of properties associated with a single device entry. For further information, see “Service Limitations” (page 434). ▲

Operating-System Services

A standard set of operating-system utilities is provided in the Driver Services Library. These services include

- `SysDebug` and `SysDebugStr`
- `PBEnqueue` and `PBDequeue`
- C and Pascal string manipulation routines
- Memory copying and clearing

For a more complete set of driver services, see Chapter 11, “Driver Services Library.”

Timing Services

The Time Manager calls `InsTime`, `PrimeTime`, and `RmvTime` have been replaced with a new set of services, described in “Timing Services” (page 416). The timing routines available are

- `SetInterruptTimer`
- `CancelTimer`
- `UpTime`
- `TimeBaseInfo`

In the past, there have been special services provided to 68K drivers to allow for delayed processing. These mechanisms, such as `dNeedTime`, `drvrDelay`, and `accRun`, are specific to the Macintosh Toolbox and the Process Manager. These facilities will continue to be provided for Toolbox code resources; drivers written to be compatible with the native driver specification will never run in a Toolbox context and hence may not make use of these facilities.

Memory Management Services

Native drivers may not call Toolbox memory management routines, particularly

- `NewPtr`
- `NewPtrSys`
- `NewHandle`
- `SetZone`

Memory allocation requests should use either a device family–specific allocation mechanism or the new memory management services. The memory management allocation and deallocation routines are

- `PoolAllocateResident`
- `PoolDeallocate`

An example of a family specific allocation mechanism is `allocb` for STREAMS drivers. `allocb` is an exported allocation mechanism provided to all STREAMS drivers and protocol modules. `allocb` uses the appropriate memory management services to its underlying operating system.

The Macintosh native driver memory management services are listed and described in Chapter 11, "Driver Services Library."

Hardware Interrupt Mechanisms

To install an interrupt handler, you use `InstallInterruptFunctions`, which replaces the earlier Slot Manager routine `SIntInstall`.

The interrupt set ID and interrupt member number values are available as `driver-ist` properties associated with each device entry in the Name Registry. For a complete discussion of native driver interrupt handling, see "Interrupt Management" (page 381).

Secondary Interrupt Services

The Deferred Task Manager call `DTInstall` is replaced by `QueueSecondaryInterruptHandler` and `CallSecondaryInterruptHandler2`. These routines are discussed in "Secondary Interrupt Handlers" (page 409).

The Deferred Task Manager maintains a queue of deferred tasks that run after hardware interrupts but before the return to the application level. The new mechanisms allow a deferred task, now called a *secondary interrupt handler,* to be queued or run on the fly. The operating-system mechanisms used to manage secondary interrupts are no longer visible to clients of the scheduling routines. The deferred task structure itself is no longer allocated by the requesting application.

Device Configuration

All device configuration information is stored in the Name Registry. All resources required by the driver will be provided to device drivers in a family-specific way or through the Name Registry. Device driver writers must follow these rules:

- Do not use the Resource Manager.
- Do not use the file system.
- Do not use the PRAM utilities.

Support for these mechanisms is not available to drivers after the first generation of Power Macintosh computers. The Name Registry provides two kinds of persistent storage; see Chapter 10, “Name Registry,” for details on how these facilities are used. In short and in general, do not use the Macintosh Toolbox from main driver code.

All information required by device drivers is located in the Name Registry. Native driver initialization routines are passed a Name Registry node pointer that identifies the corresponding device. The Name Registry programming interface provides access routines to the interesting properties required by devices. See “Standard Properties” (page 322), for names and values of properties of interest to PCI drivers for use with Mac OS.

Native drivers should not make calls to, or expect data from, the Resource Manager. There are two reasons for this rule:

- The Resource Manager is an application service, not a system service.
- Information stored in resources is unwieldy because it is impossible to distinguish code from data resources in a paging-protected or memory-protected way.

Configuration data must be supplied by the expert controlling the device or stored as property data in the Name Registry.

C H A P T E R 8

Writing Native Drivers

This chapter tells you about Macintosh native run-time drivers in the second generation of Power Macintosh computers. It covers the following subjects:

- how generic native drivers interact with the Device Manager
- how native drivers operate concurrently
- the context in which driver code is executed
- how to write a native device driver
- the Driver Loader Library
- finding, initializing, and replacing drivers
- migrating a 68K driver to the native driver environment

You need to understand the information in this chapter if you are going to write or adapt a driver to work with Mac OS. This chapter assumes that you are generally familiar with programming Power Macintosh computers, particularly with using the Device Manager and the Code Fragment Manager.

Note
The discussions of the Device Manager and the Driver Loader Library in this chapter apply only to generic drivers. For a description of generic drivers, see “Generic Driver Framework” (page 153). ◆

Native Driver Framework

All native (PowerPC) device drivers are Code Fragment Manager (CFM) fragments with the following general features:

- CFM container format
- CFM programming interfaces exported from the driver to Mac OS
- CFM programming interfaces imported by the driver from Mac OS

Generic drivers are CFM fragments that work with the Device Manager and the Driver Loader Library; family drivers are CFM fragments that use other mechanisms. Generic and family drivers are distinguished in “Generic and Family Drivers” (page 151). The general characteristics of both kinds of native drivers are briefly summarized in the next sections.

Native Driver CFM Container Format

The Code Fragment Manager CFM format provides the mechanisms for drivers to import and export data and code symbols, is integrated with Mac OS, and is documented in *Inside Macintosh: PowerPC System Software.*

Native Driver Data Exports

All native drivers, both generic and family, must export a single data symbol that characterizes the driver's functionality and origin. This symbol, called `TheDriverDescription`, must be exported as a CFM symbol. It is documented in "Driver Description Structure" (page 198).

The driver description information helps match drivers with devices. This includes letting the Device Manager or device Family Expert pick the best driver among multiple candidates. The Device Manager uses the information in the driver description to determine whether a disk-based device driver is a better match for a device than a device driver that is in the Mac OS ROM. For example, a USB class driver in the Mac OS ROM for the mouse may be a valid match for some generic aspects of a third-party mouse device. However, the driver description for the third-party device driver installed on the disk should provide vendor ID, product ID, and other data to ensure the USB Manager/ Family Expert replaces the Mac OS ROM driver when the third-party mouse is detected.

Generic Native Driver Code Exports

Previous Macintosh drivers exported five callable routines: `Open`, `Close`, `Prime`, `Control`, and `Status`. Generic native device drivers export a single code entry point, called `DoDriverIO`, that handles all Device Manager operations. It is a selector-based entry point with command codes that specify the I/O action to be performed. The device driver can determine the nature of the I/O request from the command code (`kInitializeCommand`, `kFinalizeCommand`, `kOpenCommand`, `kCloseCommand`, `kReadCommand`, `kWriteCommand`, `kControlCommand`, `kStatusCommand`, `kKillIOCommand`, `kReplaceCommand`, or `kSupersededCommand`) and command kind (`kIOSynchronousKind`, `kIOAsynchronousKind`, or `kIOImmediateKind`). `DoDriverIO` is described in "DoDriverIO Entry Point" (page 204).

Native Driver Imports

The CFM requires that a fragment import any libraries that it is dependent on. Generic drivers link to the Driver Services Library; family drivers may also be linked to family libraries. The linking lets the fragment's symbols be bound at load time. The Driver Services Library or a family library should be used instead of a Toolbox-based library when linking a device driver.

IMPORTANT

Native device drivers should use the CFM's import library mechanism to share code or data. With this technique, the CFM creates an import library fragment when the first driver is loaded. When another driver is loaded, it establishes a connection to the existing library, letting the two drivers share code or data. For further information about the CFM, see *Inside Macintosh: PowerPC System Software.* This book is listed in "Apple Publications" (page 26). ▲

In the past, driver imports have not always been rigidly characterized. The reason for now explicitly specifying the system entry points available to native drivers is to guarantee compatibility of drivers with future releases of Mac OS. For a further discussion, see "Driver Services Library" (page 154). See also the family-specific information in Chapters 13, 14, and 15.

Drivers for Multiple Cards

This section describes two cases where drivers have to deal with multiple cards. They are the following:

The first case in which a driver may need to support multiple instances of the same PCI expansion card. In that case, the CFM and Driver Loader Library (DLL)do all the work and you will get a separate CFM instance of the driver for each card. The CFM links instances of the native driver on the fly when the driver is loaded by the DLL.

The second case where a driver needs to support different, but related PCI expansion cards (or multiple functions on one card). In this case the driver should use the Code Fragment Manager to import a shared library for both code and data.

Follow these design guidelines:

- Put the shared library in the Extensions folder in the System Folder.
- Separate your code and data into card-specific and card-independent portions. Card-specific portions go into the driver, and card-independent portions go into the library.
- Some family experts like the OpenTransport and the USB version 1.2 or greater experts support the extended `'cfrg'` resource. The extended resource allows merging of multiple drivers files in a single `'ndrv'` file. See the documentation for the specific device family for further information about how this is done.

You can construct a driver that exports services for different families, such as both `'ndrv'` and `'otan'`, using a driver description structure with multiple service categories defined.

Note
The driver is responsible for synchronizing accesses to the shared library in such a way that it protects shared data structures. You can use DSL mechanisms to help with synchronization. ◆

The Device Manager and Generic Drivers

The Device Manager is part of the Mac OS that provides communication between applications and devices. The Device Manager calls generic device drivers; it doesn't manipulate devices directly. Generic drivers accept calls from the Device Manager and either cause device actions or respond by sending back data generated by devices. For further general information about drivers and the Device Manager, see *Inside Macintosh: Devices.*

For 68K drivers, the Device Manager's capabilities and services remain unchanged. For generic native drivers, the Device Manager has changed to support PowerPC driver code and to permit drivers to operate concurrently.

Native Driver Differences

For detailed information about constructing native device drivers, see "Writing a Generic Device Driver," later in this chapter. If you are already familiar with writing 68K device drivers, the following are highlights of the principal differences:

- A native driver receives its parameters through the single `DoDriverIO` entry point, subject to the calling conventions specified by the PowerPC run-time architecture. If a `DoDriverIo` routine is written in C, the correct behavior is guaranteed. This is a fundamental change from the way 68K drivers received parameters.
- A native driver doesn't have access to its driver control entry (DCE) in the unit table.
- `ImmediateIOCommandKind` is passed in the `ioKind` parameter to specify that a request must be executed immediately. If so, the driver must process the request completely and the result of the process must be reflected in the return value from the driver. `kInitializeCommand`, `kFinalizeCommand`, `kOpenCommand`, `kCloseCommand`, `kKillIOCommand`, `kReplaceCommand`, and `kSupersededCommand` calls are always immediate.
- If the `ioKind` parameter is either `SynchronousIOCommandKind` or `AsynchronousIOCommandKind`, the return value from the driver is ignored. The driver must call `IOCommandIsComplete` at some future time.
- The `Initialize` and `Finalize` commands are sent to the driver as its first and last commands. `Initialize` gives the driver information it needs to start up. `Finalize` informs the driver that the system needs to unload it.
- Drivers now receive all `OpenDriver` and `CloseDriver` calls, which connect the driver independently of initialization and finalization. In the past, the first (and only) `OpenDriver` and `CloseDriver` calls were used as the initialization and finalization mechanism.
- All native drivers must accept and respond to all command codes. The `Read_Enable`, `Write_Enable`, `Control_Enable`, and `Status_Enable` bits in the DCE are ignored. Native drivers must keep track of I/O permissions for each instance of multiple open actions and return error codes if the permissions are violated.
- The Device Manager processes zero-length reads and writes on behalf of the 68K driver. Native drivers must accept zero-length read and write commands and respond to them in an intelligent way without crashing.
- `KillIO` is no longer a control call; it is now its own command. For backward compatibility, the Device Manager converts `KillIO` traps into `KillIO` commands. It passes the old `csKillcode` control call (`csCode` = 1) without acting on it.

- The Code Fragment Manager sends CFM initialization and termination calls to a driver when the driver is loaded and unloaded. The CFM initialization routine, if present, will run prior to the driver being initialized by the Device Manager. It is possible that the driver will be loaded and its CFM initialization routine run even though it is never opened and, therefore, never closed. It is important that any processing done by a CFM initialization routine be undone by the CFM termination routine. The Device Manager may load a number of drivers looking for the best candidate for a particular device. Only the best driver is opened and remains loaded. All other CFM connections are closed, causing the CFM termination routine to run.
- Native drivers never jump to the `IODone` routine. To finish processing an I/O request, a generic native driver must call `IOCommandIsComplete` to notify the Device Manager that a given request has been completed.
- To determine the kind of request or kind of command, the `ioTrap` field of the old Device Manager parameter block has been replaced with routine parameters called `theCode` and `theKind`. Native drivers do not need to read or modify this field.
- A native driver must be reentrant to the extent that during any call from the driver to `IOCommandIsComplete` the driver may be reentered with another request. 68K device drivers would typically JMP to to `IODone` and therefore technically never be re-entered.
- A native device driver does not have any sort of header. It must however, export a data symbol called `TheDriverDescription`. A driver uses this data structure to give header-like information to the Device Manager. The Device Manager uses the information in `TheDriverDescription` to set the `dCtlFlags` field in the driver's DCE.
- Native drivers should never look at or modify directly the `iodesrbl` field of the Device Manager parameter block. (what's that field?)
- A native device driver cannot make use of the `dCtlEMask` and `dCtlMenu` fields of its DCE.
- If you set the `ioBuffer` field in an I/O parameter block to `NULL`, the Device Manager will not pass the buffer to a native driver (but it will not return an error either).
- Native drivers cannot be used for creating desk accessories.

IMPORTANT

Native drivers may use only those services provided by the Driver Services Library or family libraries. The Driver Services Library is described in Chapter 11. ▲

Native Driver Limitations

The ability of Mac OS to support generic native drivers does not mean that Mac OS contains a fully native I/O subsystem; at present the Device Manager still runs in 68K code. In addition, the 68K emulator can service interrupts only on 68K instruction boundaries. As a result, the performance of a native device driver may be greater or less than the performance of its 68K equivalent. At this time, Apple has made no commitment to furnish either a native version of the Device Manager or a combined native-68K version (fat version).

The discussions of generic native drivers in the previous sections apply only to drivers managed by the Device Manager. Other driver-like things, such as Apple Desktop Bus drivers, which are not managed by the Device Manager, realize no benefit from the Device Manager's concurrency features. These features are discussed in the next section.

Concurrent Generic Drivers

Previously, the Device Manager let drivers process only one request at a time. Although multiple requests could be pending for a driver, the Device Manager passed each new request to the driver only when the previous request was done.

Many clients of the present Device Manager contain work arounds that let the driver handle multiple requests concurrently. The Device Manager now lets native device drivers handle concurrent tasks more simply.

Drivers that support simultaneous requests should set the `kDriverIsConcurrent` bit of the `driverRuntime` flags word in the driver description structure. When dealing with a concurrent device, the Device Manager alters its request management as follows:

- All I/O requests it receives are immediately forwarded to the appropriate driver.

- The `drvrActive` bit (bit 7) in the `dCtlFlags` field of the device control block is never set.
- When a driver chooses to do standard Device Manager queuing, the parameter blocks corresponding to its requests are placed onto the device's request queue rooted by the `dCtlQHdr` field of the device control block.
- A driver that chooses to queue requests to an internal queue should set the `kdriverQueuesIOPB` bit in the `driverRuntime` flags word in the `DriverDescriptor` structure. This bit prevents the Device Manager from queueing the request to the DCE request queue. Drivers using the `kdriverQueuesIOPB` option bit must dequeue the I/O parameter block (IOPB) from any internal queues before calling `IOCommandIsComplete`.
- A driver must use the `IOCommandIsComplete` service to complete a request. It may not use the original `IODone` service. `IOCommandIsComplete` is described in the next section.
- A driver is responsible for ensuring that all requests have been completed prior to returning from a `Finalize` request. Once a `Finalize` request has been made to a concurrent driver, no further requests will be made to the driver until the driver has completed the `Finalize` request and the driver is again initialized.

Completing an I/O Request

To replace the `IODone` routine and its associated low memory global `jIODone`, a new routine has been added to the Device Manager called `IOCommandIsComplete`. The difference between `IODone` and `IOCommandIsComplete` is that while `IODone` initiates request completion processing for a request that is implicitly designated by the request queue head, a caller of `IOCommandIsComplete` must explicitly specify the request that is to be completed.

After a nonimmediate `IOCommandKind` command has been accepted, the driver performs the actions implied by the command and the IO parameter block contents. When the command has been processed, the driver must complete the command.

The driver must identify the command it is completing; this is done by passing the command ID to `IOCommandIsComplete`. The command ID is provided to a driver as the first parameter to its I/O entry point, as well as being stored in the IO parameter block's `ioCmdAddr` field (`ThePb` -> `ioParam.ioCmdAddr`).

As a result of a completion, the Device Manager takes several actions. If the command was performed synchronously, the I/O trap finishes. If the command was performed asynchronously, the requested I/O completion routine is invoked. The routine `IOCommandIsComplete` stores the `status` value in the I/O parameter block `result` field. It also starts the next request if the driver isn't concurrent and there is a request queued. The driver should never try to modify `result`.

Note
Under current versions of the Mac OS command ID == &param block. Your driver should not rely on this always being true. ◆

IOCommandIsComplete

`IOCommandIsComplete` lets a driver tell the Device Manager that an I/O request has been completed.

```
OSStatus IOCommandIsComplete(
                        IOCommandID ID,
                        OSErr result);
```

`ID`	Specifies the ID of a command.
`result`	The status value to place in the IO parameter block. This value must be a non-negative value.

DESCRIPTION

The parameter `ID` specifies the ID of a command being completed. The value of this ID is opaque and may be dependent on the operating system version, as discussed in the note on (page 349). The parameter `result` specifies the status value to place in the IO parameter block. The driver must make sure that the request that corresponds to `Command` is not queued internally when it calls `IOCommandIsComplete`, and it may not access the parameter block afterward.

EXECUTION CONTEXT

`IOCommandIsComplete` may be called from task level or secondary interrupt level, but not from hardware interrupt level. For a list of the execution contexts of other system routines that support native drivers, see “Service Limitations” (page 434).

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Note
The `OSStatus` type is described in “Error Returns” (page 156). ◆

Concurrent I/O Request Flow

The movement of multiple driver I/O requests from clients through the Device Manager to concurrent drivers and back again follows these steps:

1. A client issues an I/O request.
2. The request (in the form of an I/O parameter block) is passed to the Device Manager.
3. The Device Manager uses the `refNum` in the I/O parameter block to locate the appropriate driver.
4. The Device Manager checks the `kdriverQueuesIOPB` option bit. If the value of the bit is `false`, the Device Manager adds the I/O parameter block to the driver's DCE-based request queue.
5. The Device Manager invokes the driver's `DoDriverIO` entry point.
6. The driver may choose to leave the request on the DCE queue; alternately, if it is using the `kdriverQueuesIOPB` bit, the driver may put the request into a privately managed queue.
7. The driver starts the I/O action; if it is truly asynchronous, it returns to the Device Manager without calling `IOCommandIsComplete`.
8. If the client issued the request synchronously, the Device Manager waits for the completion of the request; otherwise, it returns control to the client.

9. Some time later, the driver determines (through a hardware or secondary interrupt routine) that the device I/O action has finished. At this time, the driver scans its private queue looking for the I/O parameter block representing the I/O action.
10. The driver uses the I/O parameter block `commandID` stored at (`ThePb` -> `ioParam.ioCmdAddr`) to issue an `IOCommandIsComplete` call. Drivers using the `kdriverQueuesIOPB` bit must make sure the I/O parameter block is not on any queue when calling `IOCommandIsComplete`.
11. The Device Manager places the result in the I/O parameter block.
12. If the I/O request was issued synchronously, control returns to the client. If the I/O request was issued asynchronously, the Device Manager invokes the client's completion routine (if one exists).
13. Control returns to the driver. The driver should not attempt to access the I/O parameter block after calling `IOCommandIsComplete`.

Driver Execution Contexts

This section discusses the general concepts and rules covering driver execution in Mac OS. You must understand these rules to ensure that your code will be compatible with future versions of Mac OS.

Code Execution in General

The environments in which code execution can occur are described in "Noninterrupt and Interrupt-Level Execution" (page 149) and may be summarized as follows:

- **Task level** is where applications and most other code are executed.
- **Hardware interrupt level** execution occurs as a direct result of a hardware interrupt request. The software executed at hardware interrupt level includes installable interrupt handlers for PCI and other devices as well as Apple-supplied interrupt handlers.
- **Secondary interrupt level** is similar to the deferred task environment. The secondary interrupt queue is filled with requests to execute subroutines posted for execution by hardware interrupt handlers. Secondary interrupt

handlers always execute sequentially. For synchronization purposes, code running at task level may also post secondary interrupt handlers for execution; these are processed synchronously from the perspective of the task level, but are serialized with all other secondary interrupt handlers.

IMPORTANT
Hardware interrupt handlers can nest on current versions of the Mac OS, but may not be able to in future products. ▲

Different execution levels have different restrictions. Task-level execution may make use of nearly any operating-system or Toolbox service, but secondary interrupt tasks and hardware interrupt handlers are allowed only a subset of those services. Eventhough a service may be described as callable from task level, you can’t call it from any task-level code. For example, drivers are not allowed to call the Toolbox.

Note
Some confusion in System 7 programming results from ad hoc rules governing execution contexts. In System 7, applications have one set of rules while their VBL tasks, Time Manager tasks, and I/O completion routines all have their own rules. Rules that establish when certain system services can and cannot be used are difficult to understand and are not fully established. For further discussion of this topic, see the discussion in Technote 1104, “Interrupt Safe Routines.” ◆

Driver Execution

The asynchronous I/O model requires that a generic driver’s responses to its `Read`, `Write`, `Control`, and `Status` entry points comply with the requirements of hardware interrupt level execution. This is because the Device Manager initiates requests that have been queued for the driver only after previously queued requests finish. Routine initiation and completion are both possible at the hardware interrupt level.

IMPORTANT

A driver's `Open`, `Close`, `Initialize`, `Finalize`, `Replace`, and `Superseded` entry points are always invoked at task level. This is the only opportunity that a driver has to allocate memory or use other services that are only available at the task level. For memory allocation guidelines, see "Memory Management Services" (page 349). ▲

"Service Limitations" (page 434) indicates which Mac OS services are available to drivers at hardware interrupt level and at secondary interrupt level. It is the responsibility of the driver writer to conform to these limitations. Drivers that violate the limitations will not work with future releases of Mac OS.

Writing a Generic Device Driver

This section discusses writing a generic native driver—one that can respond to Device Manager requests in the second generation of Power Macintosh computers. Although drivers may contain PowerPC assembly-language internal code, a native driver's interface should be written in C.

Before you decide to write your own device driver, you should consider whether your task can be more easily accomplished using one of the standard Macintosh drivers described in *Inside Macintosh.* In general, you should consider writing a device driver only if your hardware device or system service needs to be accessed at unpredictable times or by more than one application. For example, if you develop a new output device that you want to make available to any application, you might need to write a custom driver.

This section describes the Native Driver package and tells you how to

- create a driver description structure
- write native driver code to respond appropriately to Device Manager requests
- handle the special requirements of asynchronous I/O
- install and initialize the driver

Note
Only generic drivers interact with the Device Manager. The only part of this section that applies to family drivers is "Driver Description Structure" (page 198). ◆

Native Driver Package

The driver model in the second generation of Power Macintosh defines a new driver packaging format. This package may contain generic drivers that have the generic driver call interface or may contain device family drivers that have call interfaces specific to the device family.

The Native Driver package is a CFM code fragment. It may reside in the Macintosh ROM, in an expansion ROM, or in the data fork of a file. File-based native driver code fragments contain no resource fork and have a file type of `'ndrv'`. The DLL ignores the file's creator; by specifying a custom creator value registered with Apple, you can use this value to distinguish one driver from another. For a discussion of this technique, see "Using NVRAM to Store Name Registry Properties" (page 444).

The Native Driver package may house various types of drivers. The driver is expected to support services defined for the particular device family. One predefined driver type is a generic type and is called `'ndrv'` (not to be confused with the Native Driver file type `'ndrv'`).

The Native Driver package requires that at least one symbol be defined and exported by the CFM's export mechanism. This symbol must be named `TheDriverDescription`; it is a data structure that describes the driver's type, functionality, and characteristics.

Depending on the type of driver, additional symbols must be exported. The generic driver must export a single code entry point, `DoDriverIO`, which is passed to all driver I/O requests. `DoDriverIO` must respond to the `kOpenCommand`, `kCloseCommand`, `kReadCommand`, `kWriteCommand`, `kControlCommand`, `kStatusCommand`, `kKillIOCommand`, `kInitializeCommand`, `kFinalizeCommand`, `kReplaceCommand`, and `kSupersededCommand` commands. Native drivers must also keep track of I/O permissions for each instance of multiple open actions and return error codes if permissions are violated. Other driver types that support device families must export the symbols and entry points defined by the device family or device expert.

IMPORTANT

Native drivers must handle a new type of error return code, `OSStatus`. This data type is described in "Error Returns" (page 156). ▲

Driver Description Structure

The structure `DriverDescription` is used to match drivers with devices, set up and maintain a driver's run-time environment, and declare a driver's supported services.

```
struct DriverDescription {
                    OSType driverDescSignature;
                    DriverDescVersion driverDescVersion;
                    DriverType driverType;
                    DriverOSRuntime driverOSRuntimeInfo;
                    DriverOSService driverServices;
                    };

typedef struct DriverDescription DriverDescription;
typedef struct DriverDescription *DriverDescriptionPtr;

enum {kTheDescriptionSignature = 'mtej' /*first long word of
                    DriverDescription*/
                    };

typedef UInt32 DriverDescVersion;
                    enum {
                    kInitialDriverDescriptor = 0  /*first version of
                    DriverDescription*/
                    };
```

Field descriptions

`driverDescSignature`	Signature of this `DriverDescription` structure; currently `'mtej'` is used for all. Replace with the constant `kTheDescriptionSignature`
`driverDescVersion`	Version of this driver description structure, used to distinguish different versions of `DriverDescription` that have the same `driverDescSignature` value.

`driverType`	Structure that contains driver name and version.
`driverOSRuntimeInfo`	Structure that contains driver run-time information, which determines how a driver is handled when Mac OS finds it. This structure also provides the driver's name to Mac OS and specifies the driver's ability to support concurrent requests.
`driverServices`	Structure used to declare the driver's supported programming interfaces.

The `driverType`, `driverOSRuntimeInfo`, and `driverServices` structures are described in the next sections. A typical driver description is shown in Listing 8-1.

Listing 8-1 Typical driver description

```
DriverDescription TheDriverDescription =
{
    // signature info
        kTheDescriptionSignature,           // signature always first
        kInitialDriverDescriptor,           // version second

    // type info
    {
        "\pAAPL,Viper",                     // device's name (must match
                                            // name in Name Registry)
        0x1,0x0,0x40,0x2,                   // Rev 1.0.0a2
    },

    // OS run-time requirements
    {
        kDriverIsUnderExpertControl         // run-time options
        | kDriverIsOpenedUponLoad,
        "\p.Display_Video_Apple_Viper",
    },

    // OS run-time info
    {
        1,                              // number of service categories
        {
            kServiceCategoryNdrvDriver,     // we support 'ndrv' categor
            kNdrvTypeIsVideo,               // video type

    // Version of service
            1, 0, 0, 0                      // major, minor, stage, rev
        }
    }
};
```

Driver Type Structure

The `DriverType` structure contains name and version information about a driver, which is used to match the driver to a specific device. For further information about driver matching, see "Matching Drivers With Devices" (page 164).

```
struct DriverType {
                     Str31 nameInfoStr;
                     NumVersion version;
                     }

typedef UInt32      DeviceTypeMember;

typedef struct      DriverType DriverType;

typede struct       DriverType *DriverTypePtr;
```

Field descriptions

`nameInfoStr`	Name used to identify the driver and distinguish between various versions of the driver when an expert is searching for drivers. This string of type `Str31` is used to match the `name` property in the Name Registry.
`version`	Version resource used to obtain the newest driver when several identically named drivers (that is, drivers with the same value of `nameInfoStr`) are available.

Driver Run-Time Structure

The `DriverOSRuntime` structure contains information that controls how the driver is used at run time.

```
struct DriverOSRuntime {
                     RuntimeOptions driverRuntime;
                     Str31 driverName;
                     UInt32 driverDescReserved[8];
                     };

typedef OptionBits RuntimeOptions;

typedef struct DriverOSRuntime DriverOSRuntime;

typedef struct DriverOSRuntime *DriverOSRuntimePtr;
                     enum { /*DriverOSRuntime bit constants*/
                     kDriverIsLoadedUponDiscovery = 1,  /*auto-load
                       driverwhen discovered*/
                     kDriverIsOpenedUponLoad = 2, /*auto-open driver when
                       it is loaded*/
                     kDriverIsUnderExpertControl = 4, /*I/O expert
                       handles loads and opens*/
                     kDriverIsConcurrent = 8,/*supports concurrent
                       requests*/
                     kDriverQueuesIOPB = 0x10 /*Device Manager shouldn't
                       queue IOPB*/
                     };
```

Field descriptions

`driverRuntime` Options used to determine run-time behavior of the driver. The bits in this field have these meanings:

Bit	Meaning
0	System loads driver when driver is discovered.
1	System opens driver when driver is loaded.

2	Device family expert handles driver loads and opens.
3	Driver is capable of handling concurrent requests
4	The Device Manager should not queue the I/O parameter block (IOPB) in the DCE before calling the driver.

`driverName` Driver name used by Mac OS if driver type is `ndrv`. Mac OS copies this name to the area pointed to by the `dNamePtr` field of the DCE. This field is unused for other driver types.

`driverDescReserved`
Reserved for future use. Set to 0.

Driver Services Structure

The `DriverOSService` structure describes the services supported by the driver that are available to other software. Each device family has a particular set of required and supported services. A driver may support more than one set of services. In such cases, `nServices` should be set to indicate the number of different sets of services that the driver supports.

```
struct DriverOSService {
                     ServiceCount nServices;
                     DriverServiceInfo service[1];
                     };

typedef UInt32 ServiceCount;

typedef struct DriverOSService DriverOSService;

typedef struct DriverOSService *DriverOSServicePtr;
```

Field descriptions

`nServices` The number of services supported by this driver. This field is used to determine the size of the service array that follows.

`service` An array of `DriverServiceInfo` structures that specifies the supported programming interface sets.

Driver Services Information Structure

The `DriverServiceInfo` structure describes the category, type, and version of a driver's programming interface services.

```
struct DriverServiceInfo {
                     OSType serviceCategory;
                     OSType serviceType;
                     NumVersion serviceVersion;
                     };

typedef struct DriverServiceInfo DriverServiceInfo;

typedef struct DriverServiceInfo *DriverServiceInfoPtr;

enum {              /*used in serviceCategory*/
                     kServiceCategoryDisplay = 'disp', /*display*/
                     kServiceCategoryOpentransport = 'otan',/*Open
                       Transport*/
                     kServiceCategoryBlockstorage = 'blok', /*block
                       storage*/
                     kServiceCategorySCSISim = 'scsi', /*SCSI SIM*/
                     kServiceCategoryndrvdriver = 'ndrv' /*generic*/
                     };
```

Note
Current display devices use the generic device type `'ndrv'`. ◆

Field descriptions

`serviceCategory` Specifies driver support services for given device family. The following device families are currently defined:

Name	Supports services defined for
`'blok'`	block drivers family
`'disp'`	video display family
`'ndrv'`	generic native driver devices
`'otan'`	Open Transport
`'scsi'`	SCSI Interface Module

`serviceType`	Subcategory (meaningful only in a given service category).
`serviceVersion`	Version resource (`'vers'`) used to specify the version of a set of services. It lets interfaces be modified over time.

DoDriverIO Entry Point

Generic `'ndrv'` drivers must provide a single code entry point `DoDriverIO`, which responds to `kOpenCommand`, `kCloseCommand`, `kReadCommand`, `kWriteCommand`, `kControlCommand`, `kStatusCommand`, `kKillIOCommand`, `kInitializeCommand`, `kFinalizeCommand`, `kReplaceCommand`, and `kSupersededCommand` commands.

```
OSErr DoDriverIO (
                    AddressSpaceID spaceID,
                    IOCommandID ID,
                    IOCommandContents contents,
                    IOCommandCode code,
                    IOCommandKind kind);

typedef KernelID AddressSpaceID;
```

`spaceID`	The address space containing the buffer to be prepared. Mac OS 7.5 and later provide only one address space (`kCurrentAddressSpaceID`), which it automatically passes to native drivers.
`ID`	Command ID.
`contents`	An `IOCommandContents` I/O parameter block. Use the `InitializationInfo` union member when calling to initialize the driver, `FinalizationInfo` when removing the driver, `DriverReplaceInfo` when replacing, `DriverSupersededInfo` when superseding, and `ParmBlkPtr` for all other I/O actions.
`code`	Selector used to determine I/O actions.

`kind` Options used to determine how I/O actions are performed. The bits in this field have these meanings:

Bit	Meaning
0	synchronous I/O
1	asynchronous I/O
2	immediate I/O

DoDriverIO Parameter Data Structures

The data types and structures that the `DoDriverIO` entry point uses have the following declarations:

```
typedef struct OpaqueRef *KernelID;

enum{
      kInvalidID = 0
      };
      typedef KernelID IOCommandID;
```

Type `KernelID` is a 32-bit opaque identifier used to identify various operating system resources. Mac OS I/O services that create or allocate a resource return an ID. The ID is later used to specify the resource to perform operations on it or delete it. With type `OpaqueRef`, the value of the ID tells you nothing—you can't tell which resource it identifies without calling Mac OS. You also can't tell what ID you'll get back the next time you create a resource, and you can't tell the relationship between any two resources by the relationship between their IDs. When a resource is deleted, its ID becomes invalid for a long time. If you accidentally use an ID for a resource that has been deleted, chances are you'll get an error instead of accessing a different resource.

```
union IOCommandContents { /* contents are command specific*/
                     ParmBlkPtr pb;
                     DriverInitInfoPtr initialInfo;
                     DriverFinalInfoPtr finalInfo;
```

```
                    DriverReplaceInfoPtr replaceInfo;
                    DriverSupersededInfoPtr supersededInfo;
                    };

typedef union IOCommandContents IOCommandContents;

typedef UInt32 IOCommandCode;

enum{ /*'ndrv' driver services*/
      kOpenCommand, /*open command*/
      kCloseCommand, /*close command*/
      kReadCommand, /*read command*/
      kWriteCommand, /*write command*/
      kControlCommand, /*control command*/
      kStatusCommand, /*status command*/
      kKillIOCommand, /*kill I/O command*/
      kInitializeCommand, /*initialize command*/
      kFinalizeCommand, /*finalize command*/
      kReplaceCommand, /*replace driver command*/
      kSupersededCommand /*driver superseded command*/
      };

typedef UInt32 IOCommandKind;

enum{
      kSynchronousIOCommandKind = 1,
      kAsynchronousIOCommandKind = 2,
      kImmediateIOCommandKind = 4
      };

struct DriverInitInfo {
                    DriverRefNum refNum;
                    RegEntryID deviceEntry;
                    };

struct DriverFinalInfo {
                    DriverRefNum refNum;
                    RegEntryID deviceEntry;
                    };
```

```
typedef struct DriverInitInfo DriverInitInfo, *DriverInitInfoPtr;

typedef struct DriverInitInfo DriverReplaceInfo,
                     *DriverReplaceInfoPtr;

typedef struct DriverFinalInfo DriverFinalInfo,
                     *DriverFinalInfoPtr;

typedef struct DriverFinalInfo DriverSupersededInfo,
                     *DriverSupersededInfoPtr;

struct InitializationInfo {
                     refNum refNum;
                     RegEntryID deviceEntry;
                     };

struct FinalizationInfo {
                     refNum refNum;
                     RegEntryID deviceEntry;
                     };

typedef struct InitializationInfo InitializationInfo;

typedef struct InitializationInfo *InitializationInfoPtr;

typedef struct FinalizationInfo FinalizationInfo;

typedef struct FinalizationInfo *FinalizationInfoPtr;
```

Sample Handler Framework

A typical driver code framework for responding to `DoDriverIO` is shown in Listing 8-2.

Listing 8-2 Driver handler for `DoDriverIO`

```
OSErr
DoDriverIO( AddressSpaceID      spaceID,
            IOCommandID         theID,
            IOCommandContents   theContents,
            IOCommandCode       theCode,
            IOCommandKind       theKind )
{
   OSErr   result;

   switch ( theCode )
   {
       case    kInitializeCommand:
       case    kReplaceCommand:
               result = DoInitializeCmd
                         ( theContents.initialInfo->refNum,
                           &theContents.initialInfo->deviceEntry);
               break;
       case    kFinalizeCommand:
       case    kSupersededCommand:
               result = DoFinalizeCmd
                          ( theContents.finalInfo->refNum,
                            &theContents.finalInfo->deviceEntry);
               break;

       case    kOpenCommand:
               result = DoOpenCmd          ( theContents.pb );
               break;
       case    kCloseCommand:
               result = DoCloseCmd         ( theContents.pb );
               break;
       case    kKillIOCommand:
               result = DoKillIOCmd        ( theContents.pb );
               break;

       case    kReadCommand:
               result = DoReadCmd          ( theContents.pb );
               break;
       case    kWriteCommand:
               result = DoWriteCmd         ( theContents.pb );
               break;

       case    kControlCommand:
               result = DoControlCmd       ( theContents.pb );
               break;
       case    kStatusCommand:
```

```
                result = DoStatusCmd        ( theContents.pb );
                break;
        default:
                result = paramErr;
                break;
    }

    // if an immediate command make sure result = a valid result
     if ((ioCommandKind & kImmediateIOCommandKind) != 0) {
        return (result);/* immediate commands return the operation
                            status   */
    }
    else if (status == kIOBusyStatus) {
        /*
         * An asynchronous operation is in progress. The driver
           * handler promises to call IOCommandIsComplete when the
           * operation concludes.
         */
        return (noErr);
    }
    else {
            /*
            * Normal command that completed synchronously. Complete
            * the operation and return.
            */
        return (IOCommandIsComplete(ioCommandID, status));
    }
```

Getting Command Information

Any command in progress that the Device Manager has sent to a native driver can be examined using `GetIOCommandInfo`.

GetIOCommandInfo

```
OSErr GetIOCommandInfo(
                     IOCommandID ID,
                     IOCommandContents *contents,
                     IOCommandCode *command,
                     IOCommandKind *kind);
```

`ID`	Command ID.
`contents`	Pointer to the I/O parameter block or Initialize/Finalize contents.
`command`	Command code.
`kind`	Command kind (synchronous, asynchronous, or immediate).

DESCRIPTION

`GetIOCommandInfo` returns information about the active native driver I/O command identified by `ID`. `GetIOCommandInfo` will not work after a driver has completed a request.

EXECUTION CONTEXT

`GetIOCommandInfo` may be called from task level or secondary interrupt level, but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Responding to Device Manager Requests

As explained in "Generic Native Driver Code Exports" (page 185), native drivers respond to Device Manager requests by handling a single call, `DoDriverIO`. Native drivers must also be aware of what kind of request can be made, and at what execution level the request can be made. Table 8-1 shows the kind and execution levels for Device Manager requests.

Table 8-1 Kind and execution levels for Device Manager requests

Request command name	Kind of request supported	Execution level for request
`kInitializeCommand`	immediate	task
`kFinalizeCommand`	immediate	task
`kOpenCommand`	immediate	task
`kCloseCommand`	immediate	task
`kReplaceCommand`	immediate	task
`kSupercededCommand`	immediate	task
`kReadCommand`	immediate, asynchronous, or synchronous	task, hardware interrupt, or secondary interrupt
`kWriteCommand`	immediate, asynchronous, or synchronous	task, hardware interrupt, or secondary interrupt
`kControlCommand`	immediate, asynchronous, or synchronous	task, hardware interrupt, or secondary interrupt
`kStatusCommand`	immediate, asynchronous, or synchronous	task, hardware interrupt, or secondary interrupt
`kKillIOCommand`	immediate	task, hardware interrupt, or secondary interrupt

The `DoDriverIO` call interface is described in the previous section. The following sections discuss some of the internal routines a driver needs to service `DoDriverIO` requests.

Initialization and Finalization Routines

The Device Manager sends `kInitializeCommand` and `kFinalizeCommand` commands to a native driver as its first and last commands. The `kInitializeCommand` command gives the driver startup information; the `kFinalizeCommand` command informs the driver that the system would like to unload it.

A typical framework for a generic driver handler for Device Manager finalization and CFM initialization and termination commands is shown in Listing 8-3.

Listing 8-3 Initialization, finalization, and termination handlers

```
refNum          gMyReferenceNumber;
RegEntryID      gMyDeviceID;

OSErr DoInitializeCommand
            ( refNum myRefNum, regEntryIDPtr myDevice )
{
    // remember our refNum and Registry entry spec
    gMyReferenceNumber = myRefNum;
    gMyDeviceID = *myDevice;
    return noErr;
}

OSErr DoFinalizeCommand
            ( refNum myRefNum, RegEntryIDPtr myDevice )
{
#pragma unused ( myRefNum , myDevice )
    return noErr;
}

CFMInitialize ()
{
    return noErr;
}
```

```
CFMTerminate ()
{
    return noErr;
}
```

The driver's initialization routine should perform the following functions:

1. Check the device’s `AAPL,address` property to see that needed resources have been allocated. The `AAPL,address` property is described in “Fast I/O Space Cycle Generation” (page 454).
2. Enable PCI memory or I/O space, or both, using the logic illustrated in Listing 8-4.

Listing 8-4 Enabling PCI spaces

```
OSErr InitPCIMemorySpace            (RegEntryIDPtr          DeviceID,
                                     LogicalAddress         addr )
{
    UInt16      cmdWord;
    OSErr       status;

    status = ExpMgrConfigReadWord (DeviceID,addr,&cmdWord );
    if ( status != noErr )
        return status;

    cmdWord |=      cwCommandEnableMemorySpace |
                    cwCommandEnableIOSpace;

    return ExpMgrConfigWriteWord (DeviceID,addr,cmdWord );
}
```

3. Probe the device to verify the driver′s match to it, as illustrated in Listing 8-5.

Listing 8-5 Device probing

```
OSErr ProbePCIMemorySpace ( LogicalAddress addr )
{
    UInt8       ctest3;
    OSErr       status;

        status = DeviceProbe(
                    (void *) (((UInt32)addr) + CTEST3),
                    &ctest3,
                    k8BitAccess
                    );
    if ( status != noErr )
        return status;
}
```

The initialization code should also allocate any private storage the driver requires and place a pointer to it in its global variables. After allocating memory, the initialization routine should perform any other preparation required by the driver. If the handler fails to allocate memory for private storage, it should return an appropriate error code to notify the Device Manager that the driver did not initialize itself.

If the Open Firmware FCode in the device's expansion ROM does not furnish either a `"driver,AAPL,MacOS,PowerPC"` property or a unique `name` property, or if the driver's PCI `vendor-id` and `device-id` properties are generic, then the initialization routine must always check that the device is the correct one for the driver. If the driver has been incorrectly matched, the initialization routine must return an error code so the Device Manager can attempt to make a match. Driver matching is discussed in "Matching Drivers With Devices" (page 164). PCI `vendor-id` and `device-id` properties are discussed in Chapter 7, "Finding, Initializing, and Replacing Drivers."

The driver's finalization routine must reverse the effects of the initialization routine by releasing any memory allocated by the driver, removing interrupt handlers, and canceling outstanding timers. If the finalization routine cannot complete the finalization request, it can return an error result code. In any event, however, the driver will be removed.

If the initialization routine needs to install an interrupt handler, see the discussion in "Interrupt Management" (page 381).

Initialization, finalization, and termination calls are always immediate.

Open and Close Routines

You must provide both an open routine and a close routine for a native device driver. The current Mac OS does not require that these routines perform any specific tasks; however, the driver should keep track of open calls to match them with close calls. Open and close calls are always immediate.

Typical code for keeping track of `kOpenCommand` and `kCloseCommand` commands is shown in Listing 8-6.

Listing 8-6 Managing open and close commands

```
long gMyOpenCount;

OSErr DoOpenCommand (ParmBlkPtr thePb)
                     {
                     gMyOpenCount++;
                     return noErr;
                     }

OSErr DoCloseCommand (ParmBlkPtr thePb)
                     {
                     gMyOpenCount--;
                     return noErr;
                     }
```

Read and Write Routines

Driver read and write routines implement I/O requests. You can make read and write routines execute synchronously or asynchronously. A synchronous read or write routine must complete an entire I/O request before returning to the Device Manager; an asynchronous read or write routine can begin an I/O transaction and then return to the Device Manager before the request is complete. In this case, the I/O request continues to be executed, typically when more data is available, by other routines such as interrupt handlers or completion routines. "Handling Asynchronous I/O" (page 219) discusses how to complete an asynchronous read or write routine.

One rule you can always follow regarding synchronous and asynchronous operation is:

If your device driver can be called asynchronously and you call another device driver, you must call it asynchronously. And, if your device driver can be called asynchronously, always operate as if you are being called asynchronously.

In other words, you should not test to see whether an operation is synchronous or asynchronous, and do different things in each case.

Listing 8-7 shows a sample read routine.

Listing 8-7 Sample driver read routine

```
OSErr DoReadCommand (IOpb pb)
{
long    numBytes;
short   myErr;

numbytes = pb -> IORegCount;
    {
        /* do the read into pb -> iobuffer */
    }
return(myErr);
}
```

Control and Status Routines

Control and status routines are normally used to send and receive driver-specific information. However, you can use these routines for any kind of data transfer as long as you implement the minimum functionality described in this section. Control and status routines can execute synchronously or asynchronously, or immediate.

Listing 8-8 shows a sample control routine, `DoControlCommand`.

Listing 8-8 Sample driver control routine

```
MyDriverGlobalsPtr          gStore;

OSErr DoControlCommand (ParamBlkPtr pb)
{
    switch (pb->csCode)
    {
        case kClearAll:
            gStore->byteCount = 0;
            gStore->lastErr = 0;
            return(noErr);
        default:    /* always return controlErr for unknown csCode */
            return(controlErr);
    }
}
```

The control routine must return `controlErr` for any `csCode` values that are not supported. The status routine should work in a similar manner. The Device Manager uses the `csCode` field to specify the type of status information requested. The status routine should respond to whatever requests are appropriate for the driver and return the error code `statusErr` for any unsupported `csCode` value.

The Device Manager interprets a status request with a `csCode` value of 1 as a special case. When the Device Manager receives such a status request, it returns a handle to the driver's device control entry. The driver's status routine never receives this request.

Listing 8-9 shows a sample status routine, `DoStatusCommand`.

Listing 8-9 Sample driver status routine

```
MyDriverGlobalsPtr          gStore;

OSErr DoStatusCommand (ParamBlkPtr pb)
{
    switch (pb->csCode)
    {
        case kByteCount:
            pb->csParam[0] = gStore->byteCount;
            return(noErr);
        case kLastErr:
            pb->csParam[0] = gStore->lastErr;
            return(noErr);
        default:    /* always return statusErr for unknown csCode */
```

```
            return(statusErr);
        }
}
```

You can define driver-specific `csCode` values if necessary, as long as they are within the range 0x80 through 0x7FFF.

KillIO Routine

Native driver `killIO` routines take the following form:

```
OSErr DoKillIOCommand (ParmBlkPtr thePb)
                    /* check internal queue for request to be killed; */
                    /* if found, remove from queue and free request */
                    {
                    return noErr;
                    }
                    /* else, if no request located */
                    return abortErr;
```

`thePb`	Pointer to a Device Manager parameter block.

When the Device Manager receives a `KillIO` request, it removes the specified parameter block from the driver I/O queue. If the driver responds to any requests asynchronously, the part of the driver that completes asynchronous requests (such as an interrupt handler) might expect the parameter block for the pending request to be at the head of the queue. The Device Manager notifies the driver of `KillIO` requests so it can take the appropriate actions to stop work on any pending requests. After processing the `KillIO` call, the driver should check whether the `kImmediateIOCommandKind` bit is set in the `IOCommandKind` parameter and return the `KillIO` result to the Device Manager. Listing 8-2 shows an example in which the `DoKillIOCmd` case shows correct handling of the `killIO` routine.

Replace and Superseded Routines

Under certain conditions, it may be desirable to replace an installed driver. For example, a display card may use a temporary driver during system startup and then replace it with a better version from disk once the file system is running and initialized.

Replacing an installed driver is a two-step process. First, the driver to be replaced is requested to give up control of the device. Second, the new driver is installed and directed to take over management of the device. Two native driver commands are reserved for these tasks.

The `kSupersededCommand` selector tells the outgoing driver to begin the replacement process. The command contents are the same as with `kFinalizeCommand`. The outgoing driver should take the following actions:

- If it is a concurrent driver, it should wait for current I/O actions to finish.
- Place the device in a "quiet" state. The definition of this state is device specific, but it may involve such tasks as disabling device interrupts.
- Remove any installed interrupt handlers.
- Store the driver and the device state in the Name Registry as one or more properties attached to the device entry.
- Return `noErr` to indicate that the driver is ready to be replaced.

The `kReplaceCommand` selector tells the incoming driver to assume control of the device. The command contents are the same as those of `kInitializeCommand`. The incoming driver should take the following actions:

- Retrieve the state stored in the Name Registry and delete the properties stored by `kSupersededCommand`.
- Install interrupt handlers.
- Place the device in an active state.
- Return `noErr` to indicate that the driver is ready to be used.

Note
When replacing concurrent generic drivers, the Device Manager halts new commands until the replacement process is complete. ◆

Handling Asynchronous I/O

If you design any of your driver routines to execute asynchronously, you must provide a mechanism for the driver to complete the requests. Some examples of routines that you might use are the following:

- **Completion routines:** Completion routines are provided by Device Manager clients to let the Device Manager notify the client when an I/O process is finished.
- **Interrupt handlers:** If the driver serves a hardware device that generates interrupts, you can create an interrupt handler that responds to these interrupts. The interrupt handler must clear the source of the interrupt and return as quickly as possible. For more information about interrupts and how to install an interrupt handler, see “Interrupt Management” (page 381).

Clients of the Device Manager that make asynchronous calls should observe these guidelines when using asynchronous routines:

- Once you pass a parameter block to an asynchronous routine, it is out of your control. You should not examine or change the parameter block until the completion routine is called because you have no way of knowing the state of the parameter block.
- Do not dispose of or reuse a parameter block until the asynchronous request is completed. For example, if you declare the parameter block as a local variable, the function cannot return until the request is complete because local variables are allocated on the stack and released when a function returns.
- Use a completion routine to determine when an asynchronous routine has completed, rather than polling the `ioResult` field of the parameter block. Polling the `ioResult` field is not efficient and defeats the purpose of asynchronous operation.

Installing a Device Driver

There are ways to install a device driver, depending on where the driver code is stored and how much control you want over the installation process.

- You can store the device driver in an expansion ROM, as described in Chapter 4, “Startup and System Configuration.”
- You can store the device driver on disk in a file of type `'ndrv'` in the Extensions folder.
- Use the DLL

The first option, storing the driver in the card’s expansion ROM, is the normal practice because it gives the card autoconfiguration capabilities, as described in Chapter 4, “Startup and System Configuration.”

See Chapter 7, “Finding, Initializing, and Replacing Drivers,” for driver loading and installation details. Chapter 9, “Driver Loader Library,” provides details of the mechanisms available for installing and removing drivers that are listed in the Device Manager unit table.

Table 8-2 lists the driver unit numbers that are reserved for specific purposes.

Table 8-2 Reserved unit numbers

Unit number range	Reference number range	Purpose
0 through 11	–1 through –12	Reserved for serial, disk, AppleTalk, printer, and other drivers
12 through 31	–13 through –32	Available for desk accessories
32 through 38	–33 through –39	Available for old SCSI devices
39 through 47	–40 through –48	Reserved
48 through 127	–49 through –128	Available for PCI and other drivers

Driver Gestalt

Every device driver has a unique set of family-specific configuration and state information that it maintains. This configuration information often needs to be passed between the family expert and the device drivers it manages. To aid in this communication process, the native driver architecture provides a driver gestalt mechanism. Driver gestalt provides a common, unified mechanism for both native and 68K device drivers by which clients (such as applications) or family subsystem managers (such as the SCSI Manager or the Display Manager) can access family-specific configuration and state information about the driver.

For instance, the Start Manager uses `driverGestalt` to interrogate SCSI drivers for family-specific information to determine from which SCSI device to boot. The information communicated back to the Start Manager is family specific (specific to the SCSI Manager) and contains necessary data for system startup—SCSI bus ID, device ID, and disk partition. Each I/O subsystem defines unique `driverGestaltSelector` and `driverGestaltResponse` formats. The SCSI Manager driver gestalt formats are SCSI based, the Display Manager formats convey video information, and so on. Cross-device-family `driverGestalt` calls are not advised; for example, don't make SCSI Manager driver gestalt calls to video drivers.

Note
Support for driver gestalt is optional, but it is highly recommended. If a PCI device driver does not support driver gestalt, it may not work with some applications or in certain system configurations. ◆

For general information about the Mac OS gestalt mechanism, see *Inside Macintosh: Operating System Utilities.* The driver gestalt and Mac OS gestalt mechanisms differ in that the driver gestalt provides a way for the Mac OS to get information about device drivers and the traditional Mac OS gestalt provides a way for device drivers to get information about the Mac OS.

System gestalt for PCI-based Macintosh computers, which is different from driver gestalt, is described in "Macintosh System Gestalt" (page 334).

Supporting and Testing Driver Gestalt

`DriverGestaltOn`, `DriverGestaltOff`, and `DriverGestaltIsOn`, described in this section, let driver code and other software communicate about the driver's support for driver gestalt.

DriverGestaltOn and DriverGestaltOff

`DriverGestaltOn` and `DriverGestaltOff` let driver code indicate to other software that it does or does not support driver gestalt.

```
OSErr DriverGestaltOn(DriverRefNum refNum);
```

```
OSErr DriverGestaltOff(DriverRefNum refNum);
```

`refNum`	Unit table reference number.

DESCRIPTION

`DriverGestaltOn` and `DriverGestaltOff` set and clear bit 2 in the device control entry (DCE) flags word.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number

DriverGestaltIsOn

`DriverGestaltIsOn` lets other code test whether or not a driver supports driver gestalt.

```
Boolean DriverGestaltIsOn (DriverFlags flags);
```

`flags`	The `flags` word in the driver's DCE.

DESCRIPTION

`DriverGestaltIsOn` returns `true` if bit 2 in the DCE flags word is set, `false` otherwise.

Implementing Driver Gestalt

If a native driver has indicated support for driver gestalt, as described in the previous section, it must conform to these rules:

- It must respond to all unsupported status `csCode` values with a `statusErr` value, and to all unsupported control `csCode` values with a `controlErr` value. This rule is the most important for drivers to follow after calling `DriverGestaltOn`.
- It should be capable of closing properly and of removing vertical blanking (VBL) tasks, Time Manager tasks, drive queue elements, and so on. Drivers that can close should return `noErr` in response to `Close` requests. If it is absolutely not possible for the driver to close, it must respond with `closErr` and continue to function as if the `Close` request had not been issued.

- It must implement the `csCode` values listed in Table 8-3 and described in the rest of this section. Driver clients seeing the `DriverGestaltEnable` bit set will assume that these calls will either produce the required actions or result in a `statusErr` or `controlErr`.
- It must not use private csCodes with values lower than 128. All codes must be allocated withing the range 128 to 32767.

Note
The `kcsDriverGestalt` and `kcsDriverConfigure` codes produce the principal functionality of the native driver model. For historical reasons, setting the `DriverGestaltEnable` bit also requires that the other calls listed in Table 8-3 either be supported or return an error code. Future control or status calls for all native PCI drivers will be implemented using only selectors through `DriverGestalt` and `DriverConfigure`.

Table 8-3 Driver gestalt codes

Name	Value	Description
Status codes		
`kcsDriverGestalt`	43	General status information
`kcsGetPowerMode`	70	Returns card power mode*
`kcsReturnDeviceID`	120	Returns SCSI device ID in `csParam[0]`
Control codes		
`kcsDriverConfigure`	43	General configuration commands
`kcsSetStartupDrive`	44	Designates partition as a boot partition
`kcsSetPowerMode`	70	Sets card power mode*

* For a discussion of power modes, see “Card Power Controls” (page 469).

DCE Flags

DCE bit 2 indicates that a driver supports the driver gestalt interface defined in the next section. The complete list of DCE bits in the `dCtlflags` word is given in Table 8-4.

Table 8-4 DCE bits in dCtlflags word

Name	Value	Description
`VMImmune`	0	This bit indicates that your device driver is VM safe.
`reserved`	1	This bit is reserved.
`kmDriverGestaltEnableMask`	2	This bit is set if the driver supports the Driver Gestalt mechanism.
Native Driver	3	Set if the driver is a native driver (ndrv). The system sets this bit when it loads your native driver.
Concurrent	4	Set if the native driver supports concurrent operation. When loading a native driver, the system sets this bit based on the `kDriverIsConcurrent` field of the `driverOSRuntimeInfo.driverRuntime` field of your `DriverDescription`.
`dOpenedMask`	5	This bit is set if the driver is open.
`dRAMBasedMask`	6	(Not used with native drivers) This bit is set if the `dCtlDriver` field is a `DRVRHeaderHandle` rather than a`DRVRHeaderPtr`.
`drvrActiveMask`	7	This bit is set if the driver is currently processing a request.
`dReadEnableMask`	8	This bit is set if the driver handles `_Read` requests.
`dWriteEnableMask`	9	This bit is set if the driver handles `_Write` requests.

Table 8-4 DCE bits in dCtlflags word (continued)

Name	Value	Description
dCtlEnableMask	10	This bit is set if the driver handles _Control requests.
dStatEnableMask	11	This bit is set if the driver handles _Status requests.
dNeedGoodbyeMask	12	This bit is set if the driver needs a "goodbye" _Control call before the application heap is reinitialized.
dNeedTimeMask	13	This bit is set if the driver wants periodic SystemTask time through the "accRun" _Control call.
dNeedLockMask	14	This bit is set if the driver requires that its DCE and code be locked at all times while the driver is open.
reserved	15	This bit is reserved.

Using DriverGestalt and DriverConfigure

Status code csCode 43 (0x2B) is defined as DriverGestalt. It takes two parameters, at csParam and csParam+4, that contain a gestalt-like selector and long word output. Similarly, control csCode 43 is defined as DriverConfigure. It also takes two parameters, an OSType selector that specifies the requested operation and a long word. The parameter blocks have these structures:

```
struct DriverGestaltParam {
    QElemPtr            qLink;
    short               qType;
    short               ioTrap;
    Ptr                 ioCmdAddr;
    ProcPtr             ioCompletion;
    OSErr               ioResult;
    StringPtr           ioNamePtr;
    short               ioVRefNum;
    short               ioCRefNum;      /* refNum for I/O operation*/
    short               csCode;         /* == kDriverGestaltCode */
    OSType              driverGestaltSelector;
    UInt32                  driverGestaltResponse;      /* Could be a */
                                            /* pointer, bit field or */
                                            /* other format */

    UInt32                  driverGestaltResponse1;     /* Could be a */
                                            /* pointer, bit field or */
                                            /* other format */

    UInt32                  driverGestaltResponse2;     /* Could be a */
                                            /* pointer, bit field or */
                                            /* other format */

    UInt32                  driverGestaltResponse3;     /* Could be a */
                                            /* pointer, bit field or */
                                            /* other format */

    UInt16                  driverGestaltfiller; /* To pad out to the */
                                                /* size of a controlPB */
};

struct DriverConfigParam {
    QElemPtr                    qLink;
    short                       qType;
    short                       ioTrap;
    Ptr                         ioCmdAddr;
    IOCompletionUPP             ioCompletion;
    OSErr                       ioResult;
    StringPtr                   ioNamePtr;
    short                       ioVRefNum;
    short                       ioCRefNum;  /* refNum for I/O operation*/
    short                       csCode;     /* == driverConfigureCode*/
    OSType                      driverConfigureSelector;
    UInt32                      driverConfigureParameter;
};
```

The `OSType` selectors for `DriverGestalt` and `DriverConfigure` are defined according to the rules of gestalt selector definition set forth in *Inside Macintosh: Operating System Utilities.* In particular, Apple reserves all four-character sequences consisting entirely of lowercase letters and nonalphabetic characters.

DriverGestalt Selectors

Currently defined selectors for the `DriverGestalt` status call are listed in Table 8-5.

Table 8-5 `DriverGestalt` selectors

Selector	Description	Response type
`'boot'`	Parameter RAM value to designate this driver/drive	`BootResponse`
'dAPI'	API support for PC Exchange	APIResponse
`'devt'`	Type of device the driver is driving	`DevTResponse`
'dics'	[call sync only] icon suite for disk driver physical drive (formerly in csCode 22)	
'ejec'	Eject options for shutdown/restart, as defined for the Shutdown Manager	EjectResponse
'flus'	Determine if disk driver supports flush and if it needs a flush.	FlushResponse
`'intf'`	Immediate location (or interface) for device	`IntfResponse`
`'lpwr'`	True if driver supports power switching	`Boolean`
'mics'	[call sync only] icon suite for disk driver media (formerly in csCode 21)	
'mnam'	Pascal string describing the disk driver (formerly in csCode 21)	
`'pmn3'`	Minimum power consumption at 3.3 V	unsigned long*
`'pmn5'`	Minimum power consumption at 5 V	unsigned long*
`'pmx3'`	Maximum power consumption at 3.3 V	unsigned long*
`'pmx5'`	Maximum power consumption at 5 V	unsigned long*
'psta'	True if device is currently in high power mode	`Boolean`
'psup'	True if driver supports power control calls	PowerResponse
`'purg'`	True if driver has purge permission	`PurgeResponse`
`'sync'`	True if driver only behaves synchronously	`SyncResponse`

Table 8-5 DriverGestalt selectors (continued)

Selector	Description	Response type
'vers'	The version number of the driver	NumVersion†
'vmop'	The disk drive's Virtual Memory options	VMOptionsResponse
'wide'	True if driver supports the ioWPosOffset for 64-bit addressing	WideResponse

* Represents power consumed in microwatts.
† The NumVersion data structure is described on (page 269).

Note
For some types of devices, DriverGestalt responses may be dependent upon fields other than the selector field. For instance, the 'boot' selector returns a startup value that identifies a particular drive in the drive queue instead of a particular device or driver. A driver handling a partitioned disk, with each HFS partition representing a separate drive, returns a result appropriate for a particular partition, as specified by drive number in the ioVRefNum field. ◆

Table lists the four character codes and corresponding constants for the DriverGestalt selectors.

Table 8-6 DriverGestalt selector four character codes and constants

Four character code selector	Selector constants
'boot'	kdgBoot
'dAPI'	kdgAPI
'devt'	kdgDeviceType
'dics'	kdgPhysDriveIconSuite
'ejec'	kdgEject
'flus'	kdgFlush
'intf'	kdgInterface

Table 8-6 DriverGestalt selector four character codes and constants (continued)

Four character code selector	Selector constants
'lpwr'	kdgSupportsSwitching
'mics'	kdgMediaIconSuite
'mnam'	kdgMediaName
'pmn3'	kdgMin3VPower
'pmn5'	kdgMin5VPower
'pmx3'	kdgMax3VPower
'pmx5'	kdgMax5VPower
'psta'	kdgInHighPower
'psup'	kdgSupportsPowerCtl
'purg'	kdgPurge
'sync'	kdgSync
'vers'	kdgVersion
'vmop'	kdgVMOptions
'wide'	kdgWide

The following response buffers are defined for some of the driver gestalt selectors listed in Table 8-5:

```
struct DriverGestaltSyncResponse
{
    Boolean behavesSynchronously;
    UInt8       pad[3]
};

struct DriverGestaltBootResponse
{
    UInt8 extDev;                       /*  packed target (upper 5 bits)
                                             LUN (lower 3 bits) */
    UInt8 partition;                    /*  partition */
    UInt8 SIMSlot;                      /*  slot */
    UInt8 SIMsRSRC;                     /*  sRsrcID */
};

struct DriverGestaltDevTResponse
{
    OSType deviceType;
};
enum {
    kdgDiskType              = 'disk',   /* standard r/w disk drive */
    kdgTapeType              = 'tape',   /* tape drive */
    kdgPrinterType           = 'prnt',   /* printer */
    kdgProcessorType         = 'proc',   /* processor */
    kdgWormType              = 'worm',   /* write-once */
    kdgCDType                = 'cdrm',   /* cd-rom drive */
    kdgFloppyType            = 'flop',   /* floppy disk drive */
    kdgScannerType           = 'scan',   /* scanner */
    kdgFileType              = 'file',   /* logical partition based on a
                                             file (drive Container) */
    kdgRemovableType         = 'rdsk'    /* removable media hard disk */
};

struct DriverGestaltIntfResponse
{
    OSType interfaceType;
};
enum {
    kdgScsiIntf              = 'scsi', /* SCSI interface */
    kdgPcmciaIntf            = 'pcmc', /* PCMCIA interface */
    kdgATAIntf               = 'ata ', /* ATA/ATAPI interface */
    kdgFireWireIntf          = 'fire', /* FireWire 1394 interface */
    kdgExtBus                = 'card' /* Card Bus interface */
};

struct DriverGestaltAPIResponse
{
    short               partitionCmds;  /* if bit 0 is nonzero, */
                                        /* supports partition */
                                        /* control and status calls */
                    /* prohibitMounting (control, kProhibitMounting) */
                    /* partitionToVRef (status, kGetPartitionStatus) */
                    /* getPartitionInfo (status, kGetPartInfo) */
    short                   unused1;        /* All the unused fields */
                                            /* should be zero */
    short                   unused2;
    short                   unused3;
    short                   unused4;
    short                   unused5;
    short                   unused6;
    short                   unused7;
    short                   unused8;
    short                   unused9;
    short                   unused10;
};

struct DriverGestaltPowerResponse {
    unsigned long       powerValue;     /* Power consumed in µWatts */
};

struct DriverGestaltFlushResponse
{
    Boolean             canFlush;       /* Return true if driver */
                                        /*  supports the kdcFlush */
                                        /* driver configure _Control */
                                        /* call */
    Boolean             needsFlush;     /* Return true if */
                                        /* driver/device has */
                                        /* data cached */
                                        /* and needs to be flushed * 
                                        /* when the disk volume */
                                        /* is flushed by the */
                                        /* File Manager */
    UInt8               pad[2];
};

/* Flags for purge permissions */
enum {
    kbCloseOk               = 0,            /* Ok to call Close */
    kbRemoveOk              = 1,            /* Ok to call RemoveDrvr */
    kbPurgeOk               = 2,            /* Ok to call DisposePtr */
    kmNoCloseNoPurge        = 0,
    kmOkCloseNoPurge        = (1 << kbCloseOk) + (1 << kbRemoveOk),
    kmOkCloseOkPurge        = (1 << kbCloseOk) + (1 << kbRemoveOk) + (1
<< kbPurgeOk)
};

struct DriverGestaltPurgeResponse
{
    UInt16          purgePermission;    /* 0 = Do not change */
                                        /* the state of the driver */
                                        /* 3 = Do Close() and */
                                        /* DrvrRemove() this driver */
                                            /* but don't deallocate */
                                            /* driver code */
                                            /* 7 = Do Close(), */
                                            /* DrvrRemove(), and */
                                            /* DisposePtr() */
    UInt16          purgeReserved;
    Ptr             purgeDriverPointer;/* pointer to the start of */
                                            /* the driver block (valid */
                                            /* only if DisposePtr */
                                            /* permission is given */
};

struct DriverGestaltEjectResponse {
    UInt32                  ejectFeatures;      /* Features field */
};

/* Flags for Ejection Features field */
enum {
    kRestartDontEject           = 0,            /* Dont Want eject */
                                                /* during Restart */
    kShutDownDontEject          = 1,            /* Dont Want eject */
                                                /* during Shutdown */
    kRestartDontEject_Mask          = 1 << kRestartDontEject,
    kShutDownDontEject_Mask         = 1 << kShutDownDontEject
};

struct DriverGestaltWideResponse
{
    Boolean supportsWide;
};
```

Using the 'boot' Selector

The 'boot' DriverGestalt status call is made both by the Startup Disk control panel when the user selects a device and by the Start Manager when the ROM is trying to match a device in the drive queue with the device specified in PRAM. The DriveNum of the device's DrvQEl is placed in the ioVRefNum field of DriverGestaltParam. In the case of a SCSI device, it is necessary to return the data in a particular format so that the startup code knows on which SCSI bus, ID, and LUN the boot device can be found. It needs this information so that it can attempt to load that driver first. A SCSI driver can return the following data:

```
biPB.scsiHBAslotNumber          -> driverGestaltBootResponse.slot
biPB.scsiSIMsRsrcID             -> driverGestaltBootResponse.sRSRC
targetID<<3 + LUN               -> driverGestaltBootResponse.extDev
partition number                -> driverGestaltBootResponse.partition
```

As shown, the disk driver can copy the values found in BusInquiry into the slot and sRSRC fields and can generate the extDev field by left-shifting the target ID by 3 bits (0 to 31 range) and adding the logical unit number (0 to 8 range). The partition field enables the selection of a single partition on a multiply partitioned device as the boot device. It is not interpreted by the ROM or the startup disk 'cdev', so the driver can choose a meaning and a value for this field. Typically the driver would enumerate the partitions laid out on a disk and return the number of the partition for the drive specified in the ioVRefNum field.

Other Control and Status Requests

This section discusses how native drivers should respond to driver gestalt control and status requests other than DriverConfigure and DriverGestalt—that is, calls with csCode values other than 43.

Startup Drive Control Request

The basic idea for both pre- and post-PCI Power Macintosh computers is that the boot partition has kPartitionIsStartup set in its pmStatus field. At startup time, the driver searches for this partition and adds its drive queue element to the drive queue before the other partitions. When the system subsequently searches for a disk to boot from, it will find this drive queue element and boot from it.

The change with the PCI-based Power Macintosh computers is that the Startup Disk control panel allows the user to choose a startup partition. When the control panel opens, it sends a `kdgBoot` Driver Gestalt request to the driver of each volume. The result from this request uniquely identifies the partition containing that volume. When the user chooses a startup volume, the control panel sends a `kcsSetStartupDrive` (csCode 44) control request to the driver. The driver responds to this by modifying the partition map to set the `kPartitionIsStartup` bit on the partition containing the volume and clear the bit on all other partitions.

Note
The startup drive mechanism is slightly different for Apple computers, such as the iMac computer, that support the NewWorld architecture. The final details were still being ironed out when this document was written. When the information is available, it will be incorporated into the appropriate sections in the documentation that discuss disk drivers and boot issues.

SetStartupDrive Control Request

The `kcsSetStartupDrive` control call (`csCode` = 44) results when a user selects a drive from the Startup Device control panel in the current version of Mac OS. It indicates to the driver that a volume controlled by that driver (that is, one with its drive number in the `ioVRefNum` field) is the chosen startup drive. This lets a specific partition selected by the user on a multiply partitioned disk be the startup volume by allowing the driver to control which partition is inserted into the drive queue first. Mass storage drivers (those that control elements in the drive queue) that set the `driverGestaltEnable` bit must implement this control request or return `controlErr`.

Disk Partition Control and Status Requests and File Exchange

The control and status requests defined in this section are primarily used by drivers to support the Apple File Exchange application, previously known as PC Exchange.

The partition information record is a structure used to store information about a partition on a device. The fields of the structure are:

`SCSIID`

If the device is connected via a SCSI interface, this field holds the SCSI Manager `DeviceIdent` of the device. If the device is connected via an ATA interface, this field holds the ATA Manager `ataDeviceID`. Devices connected via other interfaces can use whatever value makes sense to uniquely identify the device on that bus. If no value makes sense, a device must clear this field.

`physPartitionLoc`

The block number of the first block in the partition.

`partitionNumber`

The size (in blocks) of the partition.

Note
You can determine the interface used by the device issuing the `kdgInterface` Driver Gestalt query. Drivers that support File Exchange should also support this Driver Gestalt selector. For more information about the `ataDeviceID` structure, consult the ATA Device 0/1 Software Developer Guide.

GetADrive Control Call

The `kGetADrive` control call (`csCode` = 51) asks the driver to create a new drive queue element. This control call supports synchronous and asynchronous operation.

On input, `DrvQElPtr` contains the address of a drive queue element pointer. The call creates a new drive queue element based on the supplied drive queue element and places a pointer to the new drive queue element in the supplied address.

The following describes how the fields of the new drive queue element must be filled out:

Field name	Description
drive flags	(the 4 bytes prior to qLink) Inherited from the supplied drive queue element.
`qLink`	Set up when you add the drive to the drive queue using `AddDrive`.
`qType`	Inherited from the supplied drive queue element.
`dQDrive`	Must be set to a new unique drive number.

Field name	Description
`dQRefNum`	Must be set to your driver's reference number.
`dQFSID`	Inherited from the supplied drive queue element.
`dQDrvSz`	Inherited from the supplied drive queue element.
`dQDrvSz2`	Inherited from the supplied drive queue element.
partition offset	(typically held in extra bytes beyond `dQDrvSz2`) Inherited from the supplied drive queue element.

Your driver must return the new drive queue element in the memory pointed by `csParam[0..1]`. You must not post a disk inserted event for the new drive, or send the `fsmDrvQElChangedMessage` message to the File System Manager.

RegisterPartition Control Call

The `kRegisterPartition` control call (`csCode` = 50) registers a non-Macintosh partition found on a disk. This control call supports synchronous and asynchrous operation. The driver should fill in `csParam` as follows:

```
csParam[0..1] DrvQElPtr          /* The drive queue element whose */
                                 /* partition is to be changed */
(UInt32) csParam[2..3]           /* The block number of the first */
                                 /* block in the partition */
(UInt32) csParam[4..5]           /* Size of partition in blocks */
```

In response to this call, your disk driver must retarget the specified drive queue element to represent the given partition on the disk. After this call, the drive queue element must represent a partition that starts at the block specified by `csParam[2..3]` and is of the size specified by `csParam[4..5]`.

You must not post a disk inserted event for the new drive, or send the `fsmDrvQElChangedMessage` message to the File System Manager.

IMPORTANT

The effects of this call are limited to the drive queue element in memory. This call must not change the partitioning scheme on disk. ▲

ProhibitMounting Control Call

The kProhibitMounting control call (csCode = 52) prevents the mounting of a partition. This control call supports synchronous and asynchrous operation.

In response to this call, your disk driver must mark the partition specified in csParam[0..1] so that it isn't mounted at system startup. The csParam[0..1] field contains a valid partInfoRecPtr, a pointer to a partInfoRec structure that contains information about a partition:

```
typedef struct partInfoRec
{
    DeviceIdent SCSIID;                    // DeviceIdent for the device
    unsigned long physPartitionLoc;        // physical block number of
                                               beginning of partition
    unsigned long partitionNumber;         // partition number of this
                                               partition
} partInfoRec, *partInfoRecPtr;
```

Modern versions of File Exchange do not require your driver to support this call. If you decide not to support it, make sure to return controlErr.

The partition is completely determined by the fields of the partition information record, not by the ioVRefNum field of the parameter block.

IMPORTANT
The effects of this call are permanently applied to the partition map on disk.

GetPartInfo Status Call

The kGetPartInfo status call (csCode = 51) returns information about a partition in the partInfoRec structure described earlier in "ProhibitMounting Control Call." This status call supports synchronous and asynchronous operation.

In response to this call, your disk driver must place partition information about the specified drive in the partition information record pointed to by csParam[0..1].

The driver fills in the partInfoRec structure as follows:

```
*(partInfoRecPtr)csParam.SCSIID <-              /* DeviceIdent for */
                                                /* the device */
*(partInfoRecPtr)csParam.physPartitionLoc <-        /* physical block */
```

```
                                        /* number of partition start */
*(partInfoRecPtr)csParam.partitionNumber
                                /* partition number of this partition */
```

GetPartitionStatus Status Call

The `kGetPartitionStatus` status call (`csCode` = 50) retrieves the status of a partition. This status call supports synchronous and asynchronous operation.

```
(long *)csParam[0..1]               /* partInfoRecPtr for partition */
(SInt16 *)csParam[2..3]             /* address of a short for response */
```

In response to this call, the disk driver must determine whether the partition described by the partition information record pointed to by `csParam[0..1]` is mounted and return the volume reference number, `VRefNum`, of the volume in the `SInt16` pointed to by `csParam[2..3]`, or 0 if the partition is not mounted.

For SCSI and ATA devices, you can create a `partInfoRec` from scratch, for other types of devices, you have to get one back from the driver using `kGetPartInfo` and then use the returned `SCSIID` field to fill in the `SCSIID` field of your `partInfoRec`

`SCSIID` field is only valid if the driver returns `kdgScsiIntf` in response to a `kdgInterface` Driver Gestalt query.

Low Power Mode Support Calls

Control and status calls with `csCode` = 70 are optional for all drivers. Making a control call with `csCode` = 70 sets the device's power-saving mode, while a status call returns it. Information is passed in the following structure in `csParam[0]`:

```
enum {
    kcsGetPowerMode = 70 /* returns the current power mode*/
    kcsSetPowerMode = 70 /* sets the current power mode*/
};

enum {
    pmActive        = 0, /* normal operation */
    pmStandby       = 1, /* minimal energy saving state; can go active
                                  in 5 seconds */
    pmIdle          = 2, /* substantial energy savings; can go active
```

```
                                in 15 seconds */
    pmSleep         = 3  /* maximum energy savings; device may be
                                turned off */
};

struct LowPowerMode
{
    unsigned char mode;
};
```

The differences among these low power modes are the amount of energy savings and the time it takes to return to the active state. Each device driver must determine the appropriate level of energy saving support for the device that it drives. If the device can go into active state in all possible low power states within 5 seconds, it should map both `pmIdle` and `pmSleep` to `pmStandby`. If the device takes a minimum of 10 seconds to go into active state from a low power state, then it should map `pmStandby` to `pmActive`. All device drivers should support these four modes; they should never return an error because they do not support a particular mode. Low power modes that are not possible on a given device should be mapped to other appropriate modes.

For the device to become active, it is not required that the device driver get a control call telling it to make the device active. Any operation that requires the device to become active is sufficient. For example, if a hard disk driver currently has its drive in sleep mode and it gets a read call, it should automatically wake up the drive and respond to the read request. Once the drive is made active, the device driver requires a control call telling it to put the device into some other mode. It should not put the device into an inactive mode automatically unless it is managing the device's power state independently of the Mac OS Power Manager.

Drivers that support low power mode calls should return true to the `'lpwr' DriverGestalt` call listed in Table 8-5 (page 228). Drivers that do not support these calls should return false to the `'lpwr' DriverGestalt` call, return `controlErr` to the `SetPowerMode` (`csCode` = 70) control call, and return `statusErr` to the `GetPowerMode` (`csCode` = 70) status call.

Device-Specific Status Calls

This section describes two device-specific driver gestalt status calls, `ReturnDeviceID` and `GetCDDeviceInfo`.

ReturnDeviceID Status Call

A status call with a `csCode` value of 120 returns the `DeviceIdent` value for the primary SCSI device being controlled by a driver. SCSI drivers that set the `driverGestaltEnable` bit must implement this `csCode` value as described or return `statusErr`.

GetCDDeviceInfo Status Call

A status call with a `csCode` value of 121 determines the features of a particular CD-ROM drive. Before Apple's CD-ROM driver version 5.0, this was done using the `GetDriveType` status call, which returned a specific model of CD-ROM drive. This makes client code difficult to maintain since it must be modified each time a new CD-ROM drive is introduced. To alleviate this problem, the features of the device have been encoded in testable bits. An integer containing the sustained transfer rate of the drive relative to an AppleCD 150 is also included. This information is returned in the `CDDeviceCharacteristics` structure. CD-ROM drivers that set the `driverGestaltEnable` bit must either implement this `csCode` value or return `statusErr`.

```
struct CDDeviceCharacteristics
{
    UInt8               speedMajor;     /* high byte of fixed-point 
number
                                            for drive speed */
    UInt8               speedMinor;     /* low byte of "" CD 300 == 2.2,
                                            CD_SC == 1.0 etc. */
    UInt16              cdFeatures;     /* flags for features of drive */
};

enum                    /* flags for CD features field (cdFeatures) */
{
    cdPowerInject           = 0,    /* supports power inject of media */
    cdNotPowerEject         = 1,    /* no power eject of media */
    cdMute                  = 2,    /* audio channels can be muted;
                                    audio play mode = 00xxb or xx00b */
                                    /* bits 3 and 4 are reserved */
    cdLeftPlusRight         = 5,    /* left, right channels can be mixed;
                                         audio play mode = 11xxb or xx11b 
*/
                                    /* bits 6 through 9 are reserved */
    cdSCSI2                 = 10,   /* supports SCSI-2 CD-ROM cmd set */
    cdStereoVolume          = 11,   /* supports independent volume levels
                                         for each audio channel */
    cdDisconnect            = 12,   /* drive supports SCSI disconnect/
                                         reconnect */
    cdWriteOnce             = 13,   /* drive is a write/once (CD-R) type;
                                         bits 14 and 15 are reserved */

    cdPowerInjectMask               = 1 << cdPowerInject,
    cdNotPowerEjectMask             = 1 << cdNotPowerEject,
    cdMuteMask                      = 1 << cdMute,
    cdLeftPlusRightMask             = 1 << cdLeftPlusRight,
    cdSCSI2Mask                     = 1 << cdSCSI_2,
    cdStereoVolumeMask              = 1 << cdStereoVolume,
    cdDisconnectMask                = 1 << cdDisconnect,
    cdWriteOnceMask                 = 1 << cdWriteOnce
};
```

Implementing Private Control and Status Calls

If you define private Control and Status calls for communication with your device driver, you must follow certain rules to ensure reliable operation. This section outlines these rules.

Private csCode Selection

If your driver claims to supports Driver Gestalt, it must not use any csCode below 128 for a private Control or Status call. All private csCode must be allocated from the range 128 to 32767.

Synchronous Does Not Equal System Task Time

Calling a device driver synchronously does not guarantee that the driver's entry point will run at system task time. If you are defining a Control or Status call for which your driver must do something that is not interrupt safe, you must define the call to be executed immediately.

Private Calls and Virtual Memory

If your driver supports virtual memory (you can use the `kdgVMOptions` Driver Gestalt selector to indicate this), you must be careful to avoid fatal page faults when fielding private control calls. Specifically, your driver must not cause a page fault while it is fielding a queued (that is, synchronous or asynchronous) request.

The Virtual Memory Manager holds the entire ParamBlockRec passed to all queued _Read, _Write, _Control, and _Status calls. In addition, VM holds the I/O buffer (pointed to by `ioBuffer`, for length `ioReqCount`). Therefore your driver can safely access this memory without causing a fatal page fault.

The problem comes when you define a private control call whose ParamBlockRec contains a pointer to another piece of memory. If your driver accesses that memory, it may cause a page fault. If your driver supports virtual memory, that page fault will be fatal (because a page fault while any paging device is busy is fatal).

There are a number of ways to avoid this problem.

1. Always include all information inline in the parameter block. Remember that the parameter block is automatically held for you by the Virtual Memory Manager.
2. If you must include pointers in your parameter block, define your private control call interface to be called immediate. Immediate calls to a driver do not mark the driver as busy, and hence any page faults they cause will not be fatal. However, your driver must be written to support immediate calls of this kind.
3. If none of the above are suitable, you must require that your clients hold any buffers pointed to by the parameter block.

If you make a Control or Status call to a device driver which supports paging and the parameter block contains pointers to other data structures, you should hold those data structures, just to be sure.

For more background about how the Mac OS Virtual Memory Manager prevents fatal page faults, see DTS Technote 1094, “Virtual Memory Application Compatibility.”

C H A P T E R 9

Driver Loader Library

This chapter describes the **Driver Loader Library (DLL)**, a CFM shared-library extension to the Macintosh Device Manager. The DLL provides services to locate, install, and remove drivers.

IMPORTANT

Family experts and the Mac OS startup firmware are the primary clients of the DLL. It offers services that control every aspect of driver-to-device matching and driver loading and installation. Driver loading is normally an automatic process that frees drivers from having to match themselves with devices. In some situations, however, drivers may need to perform the match themselves. ▲

The installation and removal services are provided for drivers that are called through the Device Manager. Typically, these drivers are of service type `'ndrv'`. Clients that expect to call drivers through the Device Manager should utilize these services to locate the driver, load it, install it in the unit table, and remove it.

Clients of device drivers that belong to a well-defined family type (such as networking devices within Open Transport) need not use the installation and removal services, since these drivers are not callable via the Device Manager and hence do not reside in the unit table. These clients may choose to use the standard CFM services to load their drivers and may use the loader utilities to do driver matching before using the CFM.

The Driver Loader Library services provide several major functions for drivers:

- loading and memory space management
- installation in the unit table
- removal from the unit table
- providing information about installed drivers
- driver matching

Figure 9-1 shows the roles and relationships of the Device Manager, the ROM (all Macintosh system software other than the Device Manager), and the Driver Loader Library.

Figure 9-1 Position of Driver Loader Library

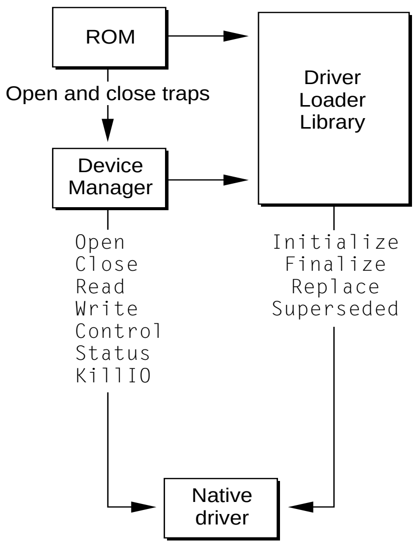
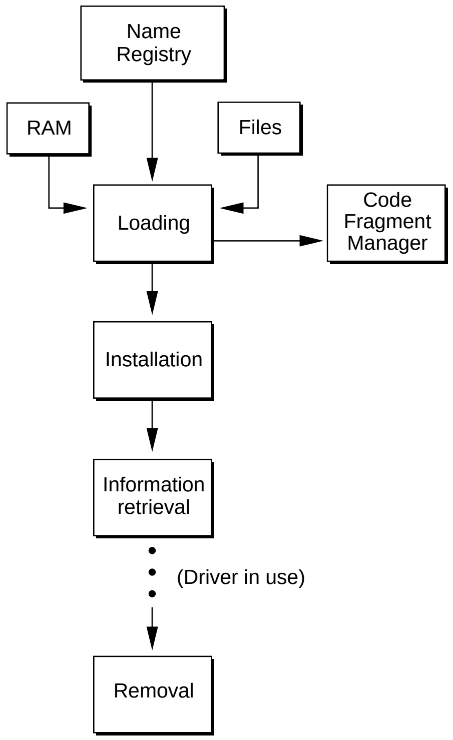


Figure 9-2 shows the relationship of the Driver Loader Library's main functions.

Figure 9-2 Driver Loader Library functions



Loading and Unloading

A driver may be loaded from any CFM container (in memory, files, or resources) as well as from a device's driver property in the Name Registry. The following services are provided for this purpose.

- `GetDriverMemoryFragment` loads a driver from a memory range.
- `GetDriverDiskFragment` loads a driver from a file.
- `FindDriverCandidates` and `ScanDriverCandidates` prepare a list of file-based drivers that potentially match a device.

- `FindDriversForDevice` finds the "best" drivers for a device, searching both ROM and disk, without making a CFM connection.
- `GetDriverForDevice` finds the "best" driver for a device and returns its CFM connection ID.
- `SetDriverClosureMemory` holds or releases a driver's memory, including any associated libraries.

The only circumstance in which `FindDriversForDevice` or `GetDriverForDevice` is required is when there is a device node in the device tree that does not have an associated driver. One instance when this might happen is if a PCI card is entered in the device tree after system startup. `FindDriversForDevice` does not create a CFM connection for the driver it finds; this service is useful if you want to browse potential drivers for a device without loading them. `GetDriverForDevice` finds the driver and creates a CFM connection for it.

The successful load of a driver yields the following results:

- a CFM `ConnectionID`
- a pointer to the driver description
- in the case of a generic native driver, a pointer to its `DoDriverIO` entry point

If the driver has a CFM initialization routine, it will be executed. The initialization routine should return `noErr` to indicate a successful load. Note that multiple drivers may be loaded in order to determine the best device-to-driver match. Therefore, a driver's CFM initialization routine should not allocate resources that cannot be released in its termination routine.

The services listed above do not affect the Device Manager's unit table. They are discussed in the next sections.

Note
Holding down the Shift, Command, N, and D keys simultaneously during Mac OS startup disables the loading of file-based drivers. ◆

GetDriverMemoryFragment

`GetDriverMemoryFragment` loads a code fragment driver from an area of memory.

```
OSErr GetDriverMemoryFragment (
                     Ptr memAddr,
                     long length,
                     ConstStr63Param fragName,
                     CFragConnectionID *fragmentConnID,
                     DriverEntryPointPtr *fragmentMain,
                     DriverDescriptionPtr *DriverDesc);
```

`memAddr`	Pointer to the beginning of the fragment in memory.
`length`	Length of the fragment in memory.
`fragName`	Optional name of the fragment (primarily used by debugger).
`fragmentConnID`	Resulting CFM `connectionID`.
`fragmentMain`	Resulting pointer to `DoDriverIO` (may be `nil`).
`DriverDesc`	Resulting pointer to `DriverDescription`.

DESCRIPTION

Given a pointer to the beginning of a driver code fragment in `memAddr` and the length of that fragment in `length`, `GetDriverMemoryFragment` loads the driver. It returns the loaded driver's CFM `connectionID` value in `fragmentConnID`, a pointer to its `DoDriverIO` entry point in `fragmentMain`, and a pointer to its driver description structure in `DriverDesc`.

Note
The CFM `connectionID` variable should be closed when the driver is unloaded. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

GetDriverDiskFragment

`GetDriverDiskFragment` loads a native driver from a file.

```
OSErr GetDriverDiskFragment(
                    FSSpecPtr fragmentSpec,
                    CFragConnectionID *fragmentConnID,
                    DriverEntryPointPtr *fragmentMain,
                    DriverDescriptionPtr driverDesc);
```

`fragmentSpec`	Pointer to a file system specification.
`fragmentConnID`	Resulting CFM `connectionID`.
`fragmentMain`	Resulting pointer to `DoDriverIO`.
`driverDesc`	Resulting pointer to `DriverDescription`.

DESCRIPTION

Given a pointer to a CFM container file system specification, `GetDriverDiskFragment` uses the CFM to find and load a driver code fragment. It returns the loaded driver's CFM `connectionID` value in `fragmentConnID`, a pointer to its `DoDriverIO` entry point in `fragmentMain`, and a pointer to its driver description in `driverDesc`.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

FindDriverCandidates

```
OSErr FindDriverCandidates (
                     RegEntryIDPtr deviceID,
                     Ptr *propBasedDriver,
                     RegPropertyValueSize *propBasedDriverSize,
                     StringPtr deviceName,
                     DriverType *propBasedDriverType,
                     Boolean *gotPropBasedDriver,
                     FileBasedDriverRecordPtr fileBasedDrivers,
                     ItemCount *nFileBasedDrivers);
```

`deviceID`	Name Registry ID of target device.
`propBasedDriver`	Address of property-based driver.
`propBasedDriverSize`	Size of property-based driver.
`deviceName`	Returned name of the device.
`propBasedDriverType`	Type of property-based driver.
`gotPropBasedDriver`	Value is `true` if property-based driver was found.
`fileBasedDrivers`	List of sorted file-based driver records.
`nFileBasedDrivers`	Count of file-based driver records.

DESCRIPTION

Given the name entry ID of a device, `FindDriverCandidates` constructs a list of file-based drivers that match the device name or one of the device-compatible names. The list is sorted from best match to least favorable match. Drivers that match the device name are listed before drivers that match a compatible name. Each of these groups are further sorted by version numbers, using the `HigherDriverVersion` service (page 269). Property-based drivers are always matched using the device name and are returned separately from file-based drivers. An I/O expert can determine a property-based driver's ranking using the `HigherDriverVersion` service.

If a `nil` list output buffer is passed, only the count of matched file-based drivers is returned. An I/O expert can call `FindDriverCandidates` first with a `nil` buffer, allocate a buffer large enough for the list, and then call `FindDriverCandidates` again with the appropriately sized buffer.

If a `nil` value is passed in `deviceID`, all drivers from the Extensions folder are returned. When using this option, pass `nil` values for all parameters except `fileBasedDrivers` and `nFileBasedDrivers`.

The list of matched drivers consists of an array of file-based driver records:

```
struct FileBasedDriverRecord {
    FSSpec          theSpec;                /* file specification*/
    DriverType      theType;                /* nameInfoStr & version
                                                number*/
    Boolean         compatibleProp;         /* true if matched using a
                                                compatible name*/
    UInt8           pad[3];                 /* alignment*/
};

typedef struct FileBasedDriverRecord
FileBasedDriverRecord,*FileBasedDriverRecordPtr;
```

A file-based driver consists of a file specification, the driver's type, and whether the driver was matched using the device name or a compatible device name.

An I/O expert can use the program logic summarized in Listing 9-1 to cycle through a list of file-based candidates.

Listing 9-1 Finding file-based driver candidates

```
FindDriverCandidates();  /* get list of candidates for a device*/
    while (Candidates in the list)
   {
      GetDriverFromFile ( FSSpec_in_Record, &driverConnectionID );
      if (InitializeThisDriver(Candidate) != NoErr))
            {
                // unhold this failed driver's memory
                //  and close its CFM connection

                UnloadTheDriver  ( driverConnectionID );

                // advance to next position in the list

             GetNextCandidate();
         }
         else
             break; // driver loaded and initialized
    }
```

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

ScanDriverCandidates

```
OSErr ScanDriverCandidates(
                     RegEntryIDPtr deviceID,
                     FileBasedDriverRecordPtr fileBasedDrivers,
                     ItemCount nFileBasedDrivers,
                     FileBasedDriverRecordPtr matchingDrivers,
                     ItemCount *nMatchingDrivers);
```

`deviceID` Name Registry ID of target device.

`fileBasedDrivers`
List of sorted file-based driver records.

`nFileBasedDrivers`
: Count of file-based driver records.

`matchingDrivers`
: File-based driver records (a subset of `fileBasedDrivers`).

`nMatchingDrivers`
: Count of driver records (<= `nFileBasedDrivers`).

DESCRIPTION

Given the name entry ID of a device and a list of `FileBasedDriverRecord` elements, `ScanDriverCandidates` constructs a list of matching file-based drivers that match the device name or one of the device-compatible names. The list is sorted from best match to least favorable match. Input to this service is an array of `FileBasedDriverRecord` elements, described in "FindDriverCandidates" (page 252). Clients can use `ScanDriverCandidates` to match drivers from a static list of candidates without having to incur the overhead of disk I/O operations.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

FindDriversForDevice

`FindDriversForDevice` finds the driver from a file and from a device tree property that represents the "best" driver for a device—that is, the latest version of the most appropriate driver, regardless of whether it is file-based or property-based. The algorithm for determining the best driver is described in "Matching Drivers With Devices" (page 164).

```
OSErr FindDriversForDevice(
                    RegEntryIDPtr device,
                    FSSpec *fragmentSpec,
                    DriverDescription *fileDriverDesc,
                    Ptr *memAddr,
                    long *length,
                    StringPtr fragName,
                    DriverDescription *memDriverDesc);
```

`device`	Device ID.
`fragmentSpec`	Pointer to a file system specification.
`fileDriverDesc`	Pointer to the driver description of driver in a file.
`memAddr`	Pointer to driver address.
`length`	Length of driver code.
`fragName`	Name of driver fragment.
`memDriverDesc`	Pointer to the driver description of driver in memory.

DESCRIPTION

Given a pointer to the `RegEntryID` value of a device, `FindDriversForDevice` finds the most suitable driver for that device. If the driver is located in a file, it returns a pointer to the driver's file system specification in `fragmentSpec` and a pointer to its driver description in `fileDriverDesc`. If the driver is a fragment located in memory, `FindDriversForDevice` returns a pointer to its address in `memAddr`, its length in `length`, its name in `fragName`, and a pointer to its driver description in `memDriverDesc`. `FindDriversForDevice` initializes all outputs to `nil` before searching for drivers.

The `fragName` parameter that `FindDriversForDevice` returns can be passed to `GetDriverMemoryFragment` (page 250) or `GetDriverDiskFragment` (page 251).

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

GetDriverForDevice

`GetDriverForDevice` loads the "best" driver for a device from memory. The algorithm for determining the best driver is described in "Matching Drivers With Devices" (page 164).

```
OSErr GetDriverForDevice(
                    RegEntryIDPtr device,
                    CFragConnectionID *fragmentConnID,
                    DriverEntryPointPtr *fragmentMain,
                    DriverDescriptionPtr *driverDesc);
```

`device`	Device ID.
`fragmentConnID`	Pointer to a fragment connection ID.
`fragmentMain`	Pointer to `DoDriverIO`.
`driverDesc`	Pointer to the driver description of driver.

DESCRIPTION

Given a pointer to the `RegEntryID` value of a device, `GetDriverForDevice` loads from memory the most suitable driver for that device. It returns the loaded driver's CFM `connectionID` value in `fragmentConnID`, a pointer to its `DoDriverIO` entry point in `fragmentMain`, and a pointer to its driver description in `driverDesc`.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

All CFM errors (See *Inside Macintosh: PowerPC System Software*)

SetDriverClosureMemory

```
OSErr SetDriverClosureMemory(
                    CFragConnectionID fragmentConnID,
                    Boolean holdDriverMemory);
```

`fragmentConnID`
: ID of driver closure (returned from other DLL loading services).

`holdDriverMemory`
: Pass `true` to hold a driver closure; `false` to free it.

DESCRIPTION

A driver and all its libraries is called a **driver closure.** When a driver is loaded and prepared for initialization by the DLL, memory for its closure is held as the final step in implementing `GetDriverMemoryFragment` and `GetDriverDiskFragment`. Closure memory is held by default to prevent page faults at primary and secondary interrupt level.

`SetDriverClosureMemory` lets you hold closure memory by setting the `holdDriverMemory` parameter to `true`. It can also be use to free memory held (unhold) for a driver closure by setting the `holdDriverMemory` parameter to `false`.

To undo the effects of `GetDriverMemoryFragment` or `GetDriverDiskFragment`, an I/O expert can call `SetDriverMemoryClosureMemory (cfmID, false)` followed by `CloseConnection (&cfmID)`. This has the effect of unloading the driver. Listing 9-2 shows a sample of code to perform this task.

Listing 9-2 Unloading a driver

```
void UnloadTheDriver ( CFragConnectionID  fragID )
{
    OSErr       Status;
    THz         theCurrentZone          = GetZone();

    // make sure the fragment is attached to the system context
    // (CFM keys context from the current heap zone)

    SetZone ( SystemZone() );

    Status = SetDriverClosureMemory (fragID,false);
    if ( Status != noErr )
      printf("Couldn't unhold pages of Driver Closure!
                (Err==%x)\n",Status);

    Status = CloseConnection(&fragID);
    if ( Status != noErr )
      printf("Couldn't close Driver Connection!
                (Err==%x)\n",Status);

    // reset the zone
    SetZone ( theCurrentZone );
}
```

Note that you must switch the current heap to the system heap before calling `CloseConnection`.

Installation

Once loaded, a driver must be installed in the unit table to become available to Device Manager clients. This process begins with a CFM fragment connection ID and results in a `refNum` value.

The installing software can specify a desired range of unit numbers in the unit table. For example, pre-version 4.3 SCSI drivers use the range 32 through 38 as a convention to their clients. If the driver cannot be installed within that range, an error is returned. The unit table may grow to accommodate the new driver, however. For the rules of this growth, see the note on (page 262).

When installing a native driver, the caller also passes the `RegEntryIDPtr` value of the device that the driver is to manage. This pointer (along with the `refNum` value) is given to the driver as a parameter in the initialization command. The driver may use this pointer to iterate through a device's property list, as an aid to initialization. The native driver should return `noErr` to indicate a successful initialization command.

These functions, described in the next sections, operate on a loaded driver fragment:

- `VerifyFragmentAsDriver` verifies fragment contents as driver.
- `InstallDriverFromFragment` places a driver fragment in the unit table.
- `InstallDriverFromDisk` places a disk-based driver in the unit table.
- `OpenInstalledDriver` opens a driver that is already installed in the unit table.

VerifyFragmentAsDriver

`VerifyFragmentAsDriver` makes sure there is a driver in a given fragment.

```
OSErr VerifyFragmentAsDriver(
                     CFragConnectionID fragmentConnID,
                     DriverEntryPointPtr fragmentMain,
                     DriverDescriptionPtr driverDesc);
```

`fragmentConnID`	CFM `connectionID`.
`fragmentMain`	Resulting pointer to `DoDriverIO`.
`driverDesc`	Resulting pointer to `DriverDescription`.

DESCRIPTION

Given a CFM `connectionID` value for a code fragment, `VerifyFragmentAsDriver` verifies that the fragment is a driver. It returns a pointer to the driver's

`DoDriverIO` entry point in `fragmentMain` and a pointer to its driver description in `driverDesc`.

RESULT CODES

`noErr`	0	No error

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

InstallDriverFromFragment

`InstallDriverFromFragment` installs a driver fragment in the unit table.

```
OSErr InstallDriverFromFragment(
                    CFragConnectionID fragmentConnID,
                    RegEntryIDPtr device,
                    UnitNumber beginningUnit,
                    UnitNumber endingUnit,
                    refNum *refNum);
```

`fragmentConnID`	CFM `connectionID`.
`device`	Pointer to Name Registry specification.
`beginningUnit`	Low unit number in unit table range.
`endingUnit`	High unit number in unit table range.
`refNum`	Resulting unit table `refNum` value.

DESCRIPTION

`InstallDriverFromFragment` installs a driver that is located in a CFM code fragment anywhere within the specified unit number range of the unit table. It invokes the driver's `Initialize` command, passing the `RegEntryIDPtr` value to it. The driver's initialization code must return `noErr` for `InstallDriverFromFragment` to complete successfully. This function returns the driver's `refNum` value but it does not open the driver.

IMPORTANT

If the unit table is filled throughout the range from `beginningUnit` to the value returned by `HighestUnitNumber` (page 272), and the table has not already grown to its maximum size, it can expand to accept the new driver. To use this feature, set `endingUnit` larger than the value returned by the `HighestUnitNumber` function. If `endingUnit` is less than or equals the returned value under these conditions, `unitTblFullErr` will be returned and the driver will not be installed. ▲

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitTblFullErr`	–29	Unit table or requested range full

Specific returns from `Initialize`, `Replace`, `Superseded`
All CFM errors (see *Inside Macintosh: PowerPC System Software*)

InstallDriverFromDisk

`InstallDriverFromDisk` locates a file in the Extensions folder, verifies that the file contains a native driver, and loads and installs the driver.

```
OSErr InstallDriverFromDisk(
                        Ptr theDriverName,
                        RegEntryIDPtr theDevice,
                        UnitNumber theBeginningUnit,
                        UnitNumber theEndingUnit,
                        DriverRefNum *theRefNum);
```

`theDriverName`	Name of a disk file containing a driver.
`theDevice`	Pointer to entry in the Name Registry.
`theBeginningUnit`	First unit table number of range acceptable for installation.
`theEndingUnit`	Last unit table number of range acceptable for installation.
`theRefNum`	Reference number returned by `InstallDriverFromDisk`.

DESCRIPTION

InstallDriverFromDisk installs a driver that is located on disk anywhere within the specified unit number range of the unit table and invokes the driver's Initialize command, passing the RegEntryIDPtr value to it. The driver's initialization code must return noErr for InstallDriverFromDisk to complete successfully. This function returns the driver's refNum value but it does not open the driver.

If the unit table is filled throughout the range from beginningUnit to the value returned by HighestUnitNumber (page 272), and the table has not already grown to its maximum size, it can expand to accept the new driver. To use this feature, set endingUnit larger than the value returned by the HighestUnitNumber function.

Note
InstallDriverFromDisk uses GetDriverMemoryFragment to load the driver, which then calls SetDriverClosureMemory to hold the driver's closure memory. ◆

RESULT CODES

noErr	0	No error
badUnitErr	–21	Bad unit number
unitTblFullErr	–29	Unit table or requested range full
fnfErr	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software)*

OpenInstalledDriver

OpenInstalledDriver opens a driver that is already installed in the unit table.

```
OSErr OpenInstalledDriver(
                    DriverRefNum refNum,
                    SInt8 ioPermission);
```

refNum	Unit table reference number.
ioPermission	I/O permission code:

fsCurPerm	0	use the current permission

`fsRdPerm`	1	allow read actions only
`fsWrPerm`	2	allow write actions only
`fsRdWrPerm`	3	allow both read and write actions

DESCRIPTION

Given an installed driver's unit table reference number, `OpenInstalledDriver` opens the driver. The Device Manager ignores the `ioPermission` parameter; it is included only to provide easy communication with the driver.

IMPORTANT

Native drivers may keep track of I/O permissions for each instance of multiple open actions and return error codes if permissions are violated. ▲

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number

Load and Install Option

Clients wishing to combine the loading and installation process in one service may want to use one of the following functions, described in the next sections:

- `InstallDriverFromFile` loads and installs a file-based driver.
- `InstallDriverFromMemory` loads and installs a memory-based driver.

InstallDriverFromFile

`InstallDriverFromFile` loads a driver from a file and installs it.

```
OSErr InstallDriverFromFile(
                    FSSpecPtr fragmentSpec,
                    RegEntryIDPtr device,
                    UnitNumber beginningUnit,
                    UnitNumber endingUnit,
                    DriverRefNum *refNum);
```

`fragmentSpec`	Pointer to a file system specification.
`device`	Pointer to Name Registry specification (may be nil).
`beginningUnit`	Low unit number in unit table range.
`endingUnit`	High unit number in unit table range.
`refNum`	Resulting unit table `refNum` value.

DESCRIPTION

`InstallDriverFromFile` installs a driver that is located on disk anywhere within the specified unit number range of the unit table and invokes the driver's `Initialize` command, passing the `RegEntryIDPtr` value to it. The driver's initialization code must return `noErr` for `InstallDriverFromFile` to complete successfully. This function returns the driver's `refNum` value but it does not open the driver.

If the unit table is filled throughout the range from `beginningUnit` to the value returned by `HighestUnitNumber` (page 272), and the table has not already grown to its maximum size, it can expand to accept the new driver. To use this feature, set `endingUnit` larger than the value retuned by the `HighestUnitNumber` function.

Note
`InstallDriverFromFile` uses `GetDriverDiskFragment` to load the driver, which then calls `SetDriverClosureMemory` to hold the driver's closure memory. ◆

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitTblFullErr`	–29	Unit table or requested range full
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

InstallDriverFromMemory

`InstallDriverFromMemory` loads a driver from a range of memory and installs it.

```
OSErr InstallDriverFromMemory (
                    Ptr memory,
                    long length,
                    ConstStr63Param fragName,
                    RegEntryIDPtr device,
                    UnitNumber beginningUnit,
                    UnitNumber endingUnit,
                    DriverRefNum *refNum);
```

`memory`	Pointer to beginning of fragment in memory.
`length`	Length of fragment in memory.
`fragName`	An optional name of the fragment (used primarily by debugger).
`device`	Pointer to Name Registry specification.
`beginningUnit`	Low unit number in unit table range.
`endingUnit`	High unit number in unit table range.
`refNum`	Resulting unit table `refNum` value.

DESCRIPTION

`InstallDriverFromMemory` installs a driver that is located in a CFM code fragment anywhere within the specified unit number range of the unit table. It invokes the driver's `Initialize` command, passing the `RegEntryIDPtr` value to it. The driver's initialization code must return `noErr` for

InstallDriverFromMemory to complete successfully. This function returns the driver's refNum value but it does not open the driver.

If the unit table is filled throughout the range from beginningUnit to the value returned by HighestUnitNumber (page 272), and the table has not already grown to its maximum size, it can expand to accept the new driver. To use this feature, set endingUnit larger than HighestUnitNumber().

Note
InstallDriverFromMemory uses GetDriverMemoryFragment to load the driver, which then calls SetDriverClosureMemory to hold the driver's closure memory. ◆

RESULT CODES

noErr	0	No error
badUnitErr	–21	Bad unit number
unitTblFullErr	–29	Unit table or requested range full
paramErr	–50	Bad parameter

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

Match, Load, and Install

Clients wishing to combine the matching of the best driver for a device, with the loading and installation process in one service, may use InstallDriverForDevice and HigherDriverVersion, described in this section. The DriverDescription data structure is used to compare a driver's functionality with a device's needs. See the discussion of the native driver container package in "Driver Loader Library" (page 245).

The Driver Loader Library picks the best driver for the device by looking for drivers in the Extensions folder and comparing those against drivers in the device's property list.

InstallDriverForDevice

`InstallDriverForDevice` installs the "best" driver for a device. The algorithm for determining the best driver is described in "Matching Drivers With Devices" (page 164).

```
OSErr InstallDriverForDevice(
                     RegEntryIDPtr device,
                     UnitNumber beginningUnit,
                     UnitNumber endingUnit,
                     DriverRefNum *refNum);
```

`device`	Pointer to Name Registry specification.
`beginningUnit`	Low unit number in unit table range.
`endingUnit`	High unit number in unit table range.
`refNum`	Resulting unit table `refNum` value.

DESCRIPTION

`InstallDriverForDevice` finds, loads, and installs the best driver for a device identified by its `RegEntryID` value. It installs the driver anywhere within the specified unit number range of the unit table and invokes its `Initialize` command, passing the `RegEntryIDPtr` value to it. The driver's initialization code must return `noErr` for `InstallDriverForDevice` to complete successfully. This function returns the driver's `refNum` value but it does not open the driver.

If the unit table is filled throughout the range from `beginningUnit` to the value returned by `HighestUnitNumber` (page 272), and the table has not already grown to its maximum size, it can expand to accept the new driver. To use this feature, set `endingUnit` larger than `HighestUnitNumber()`.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitTblFullErr`	–29	Unit table or requested range full
`fnfErr`	–43	File not found

All CFM errors (see *Inside Macintosh: PowerPC System Software*)

HigherDriverVersion

`HigherDriverVersion` compares two driver version numbers, normally the values in their `DriverDescription` structures. It returns a value that indicates which driver is the latest version. This service may be used by any software that loads or evaluates drivers.

```
short HigherDriverVersion (
                     NumVersion *driverVersion1,
                     NumVersion *driverVersion2);

struct NumVersion {
    UInt8 majorRev;                   /* 1st part of version number */
                                      /* in BCD */
    UInt8 minorAndBugRev;             /* 2nd and 3rd part of version
                                      /* number share a byte */
    UInt8 stage;                      /* stage code: dev, alpha, */
                                      /* beta, final */
    UInt8 nonRelRev;                  /* rev level of nonreleased */
                                      /* version */
};
```

`driverVersion1`
First version number being compared.

`driverVersion2`
Second version number being compared.

DESCRIPTION

`HigherDriverVersion` returns 0 if `driverVersion1` and `driverVersion2` are equal. It returns a negative number if `driverVersion1` < `driverVersion2` and a positive number greater than 0 if `driverVersion1` > `driverVersion2`. If both drivers have `stage` values of `final`, a `nonRelRev` value of 0 is evaluated as greater than any nonzero number.

Stage codes are the following:

```
developStage        = 0x20
alphaStage          = 0x40
betaStage           = 0x60
finalStage          = 0x80
```

Driver Removal

Clients wishing to remove an installed driver should use `RemoveDriver`.

RemoveDriver

`RemoveDriver` removes an installed driver.

```
OSErr RemoveDriver(
                    DriverRefNum refNum,
                    Boolean Immediate);
```

`refNum`	Reference number of driver to remove.
`Immediate`	Value of `true` means don't wait for driver to become idle.

DESCRIPTION

`RemoveDriver` accepts a `refNum` value and unloads a code fragment driver from the unit table. It invokes the driver's `Finalize` command. If called as immediate, it doesn't wait for driver to become inactive.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number

Getting Driver Information

Clients wishing to acquire information about an installed driver should use `GetDriverInformation`.

GetDriverInformation

`GetDriverInformation` returns a number of pieces of information about an installed driver.

```
OSErr GetDriverInformation(
                    DriverRefNum refNum,
                    UnitNumber *unitNum,
                    DriverFlags *flags,
                    DriverOpenCount *count,
                    StringPtr name,
                    RegEntryID *device,
                    CFragHFSLocator *driverLoadLocation,
                    CFragConnectionID *fragmentConnID,
                    DriverEntryPointPtr *fragmentMain,
                    DriverDescription *driverDesc);
```

`refNum`	Reference number of driver to examine.
`unitNum`	Resulting unit number.
`flags`	Resulting DCE flag bits.
`count`	Number of times driver has been opened.
`name`	Resulting driver name.
`device`	Resulting Name Registry device specification.

`driverLoadLocation`
: Resulting CFM fragment locator from which driver was loaded.

`fragmentConnID`
: Resulting CFM connection ID.

`fragmentMain`
: Resulting pointer to `DoDriverIO`.

`driverDesc`
: Resulting pointer to `DriverDescription`.

DESCRIPTION

Given the unit table reference number of an installed driver, `GetDriverInformation` returns the driver's unit number in `unitNum`, its DCE flags in `flags`, the number of times it has been opened in `count`, its name in `name`, its `RegEntryID` value in `device`, its CFM fragment locator in `driverLoadLocation`, its CFM connection ID in `fragmentConnID`, its `DoDriverIO` entry point in `fragmentMain`, and its driver description in `driverDesc`.

Code that calls `GetDriverInformation` must always supply an `FSSpec` file specification with the CFM locator. For an example, see Listing 9-3 (page 274).

Note
With 68K drivers, `GetDriverInformation` returns meaningful information in only the `unit`Num, `flags`, `count`, and `name` parameters. ◆

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number

Searching for Drivers

The routines described in this section help clients iterate through the unit table, locating installed drivers.

HighestUnitNumber

`HighestUnitNumber` returns the currently highest valid unit number in the unit table.

```
UnitNumber HighestUnitNumber (void);
```

DESCRIPTION

`HighestUnitNumber` takes no parameters. It returns a `UnitNumber` value that represents the highest unit number in the unit table.

LookupDrivers

`LookupDrivers` is used to iterate through the contents of the unit table.

```
OSErr LookupDrivers(
                    UnitNumber beginningUnit,
                    UnitNumber endingUnit,
                    Boolean emptyUnits,
                    ItemCount *returnedRefNums,
                    DriverRefNum *refNums);
```

`beginningUnit`	First unit in range of units to scan.
`endingUnit`	Last unit in range of units to scan.
`emptyUnits`	A value of `true` means return available units; a value of `false` means return allocated units.
`returnedRefNums`	Maximum number of reference numbers to return; on completion, contains actual number of reference numbers returned.
`refNums`	Resulting array of returned reference numbers.

DESCRIPTION

Given the first and last unit numbers to scan, `LookupDrivers` returns the reference numbers of both native and 68K drivers. The `emptyUnits` parameter tells it to return either available or allocated units, and `returnedRefNums` tells it the maximum number of reference numbers to return. When `LookupDrivers` finishes, `returnedRefNums` contains the actual number of reference numbers returned.

The sample code shown in Listing 9-3 uses `HighestUnitNumber` and `LookupDrivers` to print out the reference numbers of all installed drivers and obtain driver information.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`paramErr`	–50	Bad parameter

Listing 9-3 Using the `LookupDrivers` function

```
OSStatus FindAllDrivers (void)
{
    ItemCount           theCount        = 1;
    UnitNumber          theUnit         = 0;
    DriverRefNum        theRefNum,      *fullSizedRefNumBuffer;

    // method #1: iterate with a small output buffer

    while ( (theUnit <= HighestUnitNumber()) &&
     (LookupDrivers (theUnit, theUnit, false, &theCount, &theRefNum) ==noErr))
    {
        if (theCount == 1) printf ("Refnum #%d is allocated.\n",theRefNum);
        theCount = 1;
        theUnit++;
    }

    //  method #2: get all refnums with one call
    fullSizedRefNumBuffer = NewPtr ((HighestUnitNumber() + 1) *
     sizeof(DriverRefNum));
    theCount = (HighestUnitNumber() + 1);
    LookupDrivers (0, HighestUnitNumber(), false, &theCount,
     fullSizedRefNumBuffer);

    for(theUnit=0,theUnit <theCount;theUnit++)
    {
        printf("Refnum #%d is allocated.\n", fullSizedRefNumBuffer [theUnit]);
        ShowDriverInfo (fullSizedRefNumBuffer [theUnit]);
    }
    DisposePtr(fullSizedRefNumBuffer);
    return noErr;
}

ShowDriverInfo (DriverRefNum *refNum)
{
    UnitNumber                  theUnit;
    DriverRefNum                aRefNum;
    DriverFlags                 theFlags;
    FSSpec                      driverFileSpec;
    RegEntryID                  theDevice;
    CFragHFSLocator             theLoc;
    Str255                      theName;
    CFragConnectionID           fragmentConnID;
    DriverOpenCount             theOpenCount;
    DriverEntryPointPtr         fragmentMain;
    DriverDescription           theDriverDescription;

    theLoc.u.onDisk.fileSpec = &driverFileSpec; /* See note below */

    GetDriverInformation (          aRefNum,
                                    &theUnit,
                                    &theFlags,
                                    &theOpenCount,
                                    theName,
                                    &theDevice,
                                    &theLoc,
                                    &fragmentConnID,
                                    &fragmentMain,
                                &theDriverDescription);
    printf ("Driver's flags are:  %x\n", theFlags);
}
```

IMPORTANT
When calling `GetDriverInformation`, always supply an `FSSpec` file specification as shown in the preceding sample. Failure to do so may cause the DLL or the CFM to crash the system or overwrite the system heap. ▲

Note
You can also use the DLL to load a native driver without any associated hardware device. Just pass `nil` in `RegEntryIDPtr` to the DLL installation service. ◆

C H A P T E R 1 0

Name Registry

This chapter describes the Name Registry, a data structure maintained by Mac OS that stores hardware and software configuration information in the second generation of Power Macintosh computers.

This chapter presents general concepts followed by a detailed discussion of the Name Registry programming interface. Because native device drivers must access the Registry, developers writing new device drivers or upgrading existing drivers should read this chapter.

Concepts

People identify things by giving them names. In computer systems, names are used to identify machines, files, users, devices, and so on. The Name Registry provides a way for device drivers and system software to store names. The Registry does not store the things named, just important pieces of information about the things. The information stored is determined by the creator of the name entry and may include such data as the physical location of the thing, technical descriptions of it, and logical addresses.

Name entries are created in the Name Registry by expert software. Each expert owns specific entries and is responsible for removing them when they are no longer needed. Clients search for entries the expert has placed in the Registry, making the Registry a rendezvous point for clients and experts. The Registry does not provide general communication between clients and experts; it only helps clients and experts find each other. After clients and experts find each other, different software helps them communicate directly.

The Name Registry is similar to the name services used in some other computing environments. In concept it resembles the X.500 or BIND (named) network name services. However, the present implementation of the Name Registry is less general; it is optimized for the specific needs of hardware and device driver configuration.

The Name Tree

Name entries in the Name Registry are connected together in a tree.

Note
Code must not depend on the order in which name entries are found in the Registry. ◆

Software finds name entries in the Registry by locating the ones that it already knows about and then examining entries found nearby. By knowing to what a name entry refers, a program can find other entries that might be used for a similar or related purpose.

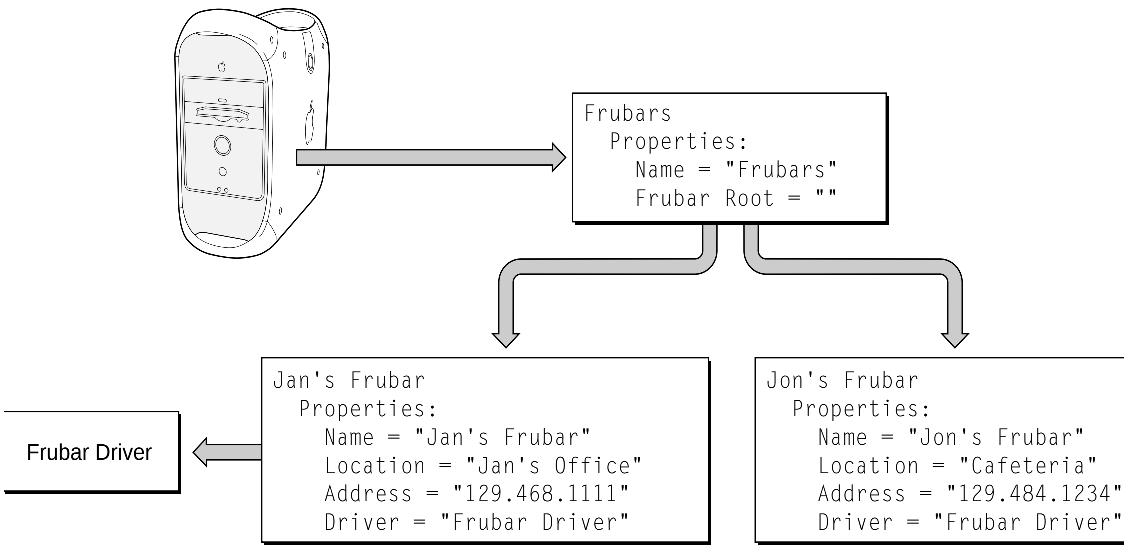
The name Registry is based on an origin entry called the *root*. All name entries in the tree may be described by a pathway through the tree starting from the root.

Name Properties

Each name entry in the Registry is accompanied by a set of properties. Each property has a name and a value. By looking at the properties associated with a name entry, software can determine what the entry identifies and what its uses are.

Software uses Registry properties to find other software. For example, if a user specifies a name while running an application, the application may look up the name in the Registry and use the properties associated with it to determine what the name represents in the system. For example, a distributed application could ask the user to choose a network interface. From the properties that accompany the name of the interface in the Registry, the application could find the device driver that controls the network interface and the parameters needed to open the network device, as diagrammed in Figure 10-1.

Figure 10-1 Using name properties



How the Registry Is Built

During system startup, the Open Firmware support code in the Macintosh ROM
creates a device tree, as described in Chapter 4, “Startup and System Configuration.” When Mac OS is launched, it extracts device information from the device tree in the following steps:

1. Search for devices.
2. Add a name entry and a set of properties to the Registry for each device.
3. Find a driver for each device.
4. Initialize the driver.

Connections between name entries are formed when the entries are added to the Registry. The connections have direction and point from an existing entry to the new one.

The Expansion Bus Manager places most of the name entries in the Registry during system startup. However, some hardware provides standard ways to probe for devices and return information describing them. In this case, the low-level expert responsible for that variety of hardware finds the devices and adds their names to the Registry. The low-level expert attaches descriptive information for each device to the name entry as properties. Low-level experts are described in "Terminology" (page 141). In a few cases, drivers may enter names and properties in the Registry directly.

The software entity that creates a name entry owns it, whether it is the Expansion Bus Manager, a low-level expert, or a device driver. Only the owner should remove a name entry. Since most device drivers do not create entries in the Registry, most drivers never remove them.

Name Registry Overview

This section summarizes the scope, design goals, limitations, and terminology of the Name Registry.

Scope

The naming services provided by the Name Registry are intended to serve local clients on a single computer only. Experts that create name entries include the low-level experts and the Expansion Bus Manager. Clients include device drivers, control panels, family experts, and other device management software.

Limitations

The Name Registry supports a relatively small number of entries. Other limitations include the following:

- Because all Registry contents reside in RAM, the number of name entries supported is limited by the available RAM space.
- Name entry creation and searching processes do not have to be fast.
- The Registry's information is volatile; information in it is lost when the system is restarted unless the information is saved to NVRAM or disk storage.

Terminology

The Name Registry uses these special terms:

- *name:* a null-terminated character string representing a thing or a concept
- *name entry:* the representation of a name in the Name Registry. Name entries are connected to form a *name tree.*
- *entry ID:* a unique ID that Mac OS gives to a name entry
- *path:* a sequence of colon-separated names
- *property:* a name-and-value pair associated with a name entry, which describes some characteristic of the thing represented by the entry
- *modifier:* hardware- or implementation-specific information associated with a name entry or property. Modifier information is stored as bits in a 32-bit word.

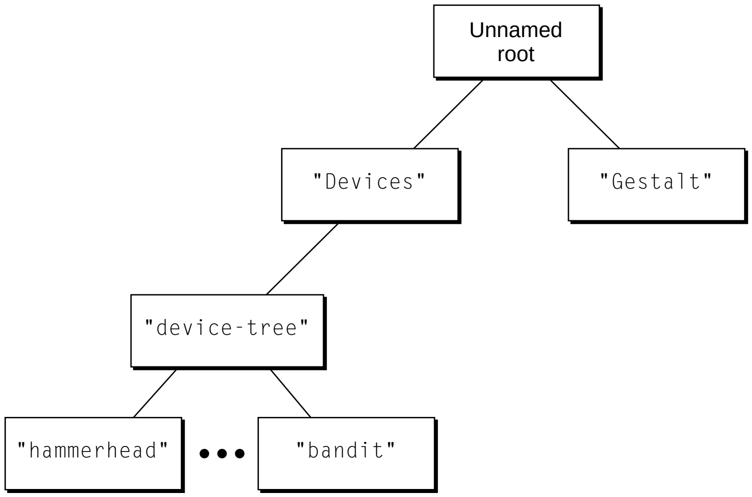
Registry Topology

The topology of the Name Registry can be summarized as follows:

- An unnamed root exists at the top of the Registry tree.
- A `Devices` name entry exists under the root. It represents the I/O universe for the computer.
- The device tree exists as a descendant (child) of the `Devices` name entry, with a new name `device-tree`, which is machine independent. This descendant represents the Power Macintosh I/O hardware.
- The `Gestalt` entry is another child of the root, making it a peer to `Devices`. This entry is not guaranteed to be available in future Macintosh computer implementations.

These relationships are diagrammed in Figure 10-2.

Figure 10-2 Typical Name Registry structure



The Device Tree

The device tree is a data structure that the Macintosh startup firmware creates in system RAM to provide information about configured devices to other software, including firmware on PCI cards. Attached to it are the drivers and support software that devices need to operate. The device tree in PCI-compatible Power Macintosh computers is similar to the sResource table previously used in NuBus-compatible Macintosh computers. For further information, see "Startup Firmware" (page 83).

The device tree is the structure from which Mac OS extracts the original information to create the device portion of the Name Registry. A device tree entry may be a device node (a entry that serves one hardware device) or a property entry (a list of name-and-value pairs associated with a device entry). Device nodes may have child device nodes, creating a branching tree structure; however, the tree begins with a single root entry. Device nodes in the single systemwide device tree may serve devices that are connected to the PowerPC processor bus through different bridges. Each device entry in the tree has one or more property nodes. An important use of property nodes is to store drivers associated with PCI card devices.

You can view the Name Registry generally as a global tree structure with a large branch equal to the original Open Firmware device tree plus and minus a few properties. When bringing the Open Firmware device tree to Mac OS through the Open Firmware client interface, the only pruning of the original tree is to delete drivers for other operating systems that may be stored there. All drivers with a `driver,AAPL,MacOS,PowerPC` property are brought into the Mac OS Name Registry.

The device tree for a pre-NewWorld PCI-based Power Macintosh computer (the Power Macintosh 9500) is shown in Listing 10-1. Note that the Bandit and Hammerhead ASICs are also shown in Figure 10-2. Listing X shows the device tree for a Power Macintosh G3 Pro.

Listing 10-1 A typical device tree

```
/bandit@F2000000
   /gc@10
     /53c94@10000
       /sd@0,0
     /mace@11000
     /escc@13020
     /escc@13000
     /awacs@14000
     /swim3@15000
     /via-cuda@16000
       /adb@0,0
         /keyboard@0,0
         /mouse@1,0
       /pram@0,0
       /rtc@0,0
       /power-mgt@0,0
     /mesh@18000
       /sd@0,0
   /bandit@B
   /AAPL,8250@E
 /bandit@F4000000
   /bandit@B
   /ATY,mach64@E
 /hammerhead@F8000000
```

Real and Virtual Devices

Name entries can be associated with many different things, including real devices and virtual devices. A virtual device is represented by a name entry for which there is no hardware. Any piece of software can add a virtual device just by creating a new entry in the `Devices` section of the Name Registry. It can mimic hardware to any degree by its selection of properties and its location in the tree topology. For example, a virtual device might enter only a logical address, using an `AAPL,address` property, or it might enter a full set of properties to mimic the behavior of a real device such as a SCSI controller.

Future versions of Mac OS will use the Name Registry to store information about many kinds of system components besides devices.

Using the Name Registry

This section describes the Name Registry programming interface available to device drivers and other device control software in the second generation of Power Macintosh computers.

Determining If the Name Registry Exists

You can use the Gestalt Manager to determine if the Name Registry exists in the user’s version of Mac OS, using the gestalt selector `'nreg'`. Check the routine’s error return first; `Gestalt` will return `gestaltUndefSelectorErr` if the Name Registry is not present. If the routine was successful, check the gestalt return for the Name Registry version number (currently 0). The Gestalt Manager is discussed in *Inside Macintosh: Operating System Utilities.* Its use in the second generation of Power Macintosh computers is described in “Macintosh System Gestalt” (page 334).

If the Name Registry is not present, the computer does not support PCI cards. The converse is not true. For example, the iMac computer does not have a PCI bus for expansion cards, but it does utilize and support the Name Registry.

PCI Bus Identification

When the user's system is running Mac OS, you can use the Name Registry to determine if a PCI bus exists in it. Use the `RegistryEntrySearch` routine, described on (page 301), to locate a name entry that has a property named `"device-type"` with a property value `"pci"`. If the routine returns `noErr` and its `done` parameter returns `false`, then a PCI bus exists.

Name Entry Management

The name graph is based on an anonymous, unnamed root entry under which all other entries live. This root does not appear in pathnames, and it can be referenced only indirectly, using `null` for its parent `entryID` value.

Given a parent `entryID` value and the pathname `:aaaa:bbbb`, `aaaa` is a child of the specified parent name entry. If the specified parent name entry is `null`, the root entry is assumed to be the parent and the path is equivalent to an absolute path.

Names for the entries just below the root (children of the root) are generic names representing categories of things such as devices, processes, volumes, and so on. As you move down the tree the things become more specific, depending on their organization within each category.

Name Entry Identifiers

Each name entry in the Name Registry is given a unique ID, of type `RegEntryID`, that code can use to reference the entry. The structure of this ID is opaque—it is accessible only to system code and may change in future releases of Mac OS. For a discussion of opaque IDs, see the note on (page 349).

Name entry identifiers might contain allocated data, so Mac OS includes operations to copy and dispose of them. See "ID Management" (page 291).

Pathnames

Name Registry paths are colon-separated lists of name components. Name components may not contain colons themselves.

Paths and name components are presented as null-terminated character strings.

Paths follow parsing rules similar to Apple file system absolute and relative pathnames. However, the Apple double colon (`::`) parent directory syntax is not currently supported.

Absolute pathnames are assumed to be rooted to the anonymous root. For example, in the pathname `aaaa:bbbb`, `aaaa` is a child of the root and `bbbb` is a child of `aaaa`. Relative pathnames are rooted to a specified parent name entry identified using an `entryID` value.

Pathnames, both absolute and relative, should not be hard coded in expert or driver code unless it is certain that the subset of the tree represented by the pathnames will remain static. The location of things in the tree can and will change over time, thus changing the pathnames. For example, a card can be inserted into one of several slots and potentially change the parent name entry that represents the slot. However, pathnames are useful for displaying the current topology of the tree or subtree or for referencing static portions of the tree.

Finding Name Registry Components

Objects in the Name Registry should be located by means of search or iterate calls using properties to identify and match the desired entry. Code can search for properties (name and value combinations) that uniquely identify the required name entry. Searching for generic names such as `"SCSI"` or `"ADB"` is not a good idea because a generic name search can find many unrelated entries.

Using Iterate Routines

Writing code to interate through a set of names consists of a call to begin the iteration, the iteration loop, and a call to end the iteration. The call to end the iteration should be made even in the case of an error, so that allocated data structures can be freed. Here is the basic code structure for traversing names in the Name Registry:

```
Create(...)
Set(...)                    // optional
do {
        Iterate(...);       // or Search(...);
} while (!done);
Dispose(...);
```

Two different name entry iterations are provided, direction oriented and search oriented. The type of iteration is indicated by the call used to retrieve the next name entry. All the Mac OS routines used are described in "Name Iteration and Searching" (page 297). Rules for direction iteration are given below; rules for search iteration are given in the next section.

- `RegistryEntryIterate`, described on (page 299), is used to traverse and explore the Name Registry. An iteration operation begins at a starting entry and moves in a direction defined by the `relationship` parameter. The relationship determines which entries relative to the starting entry are to be included in the search—children, parents, siblings, or descendants. Each iterate call returns the next entry encountered along the designated path. You can change the direction at any time by specifying a new `relationship` parameter in your next iterate call. You can continue in the current direction by specifying `kRegIterContinue` for the `relationship` parameter. Remember that the direction is relative to the last entry returned from the previous iterate call.
- When an entry iterator is created via `RegistryEntryIterateCreate`, it is initialized to the default starting entry root and with a relationship of `kRegIterDescendants`. This lets you iterate over the entire Name Registry.
- You can use `RegistryEntryIterateSet` to set the iterator to some name entry other than the root, limiting the iteration to some subset of the Name Registry. To change the default relationship, specify a new relationship as a parameter to your first iterate call.
- An iteration sequence is complete when either it finds what it is looking for or the `done` parameter returns `true`, indicating that there are no more entries in the specified direction. When `done` is `true` no error code is returned and the contents of `foundEntry` are indeterminate. The iterator must be reset, using `RegistryEntryIterateSet`, before it can be used again for a subsequent search or iterate operation.
- Each iterate call should describe the next relationship of interest.
- Don't mix iterators for iterate and search routines without reinitializing the iterator value by means of `RegistryEntryIterateSet`.

Here are some hints for using relationships while iterating:

- To iterate through all the descendants of an entry, specify `kRegIterDescendants` on the first iterate call and then specify `kRegIterContinue` until `done` is `true`.

- To iterate through the children of an entry, specify `kRegIterChildren` on the first iterate call and then specify `kRegIterContinue` until `done` is `true`.
- To iterate through the siblings of an entry, specify `kRegIterSiblings` on the first iterate call and then specify `kRegIterContinue` until `done` is `true`. Siblings do not include the current entry.
- To iterate through the parents of an entry, specify `kRegIterParents` on the first iterate call and then specify `kRegIterContinue` until `done` is `true`. Note that there is only one parent in the current implementation of the Name Registry.
- To navigate down the registry hierarchy, specify `kRegIterChildren` until you find the level you are looking for or until `done` is `true` (which indicates that you have reached the bottom). The latter case is useful when deleting a subtree, because you must delete the children before you can delete a parent.
- To navigate up the Name Registry hierarchy, specify `kRegIterParents` until you find the level you are looking for or until `done` is `true` (which indicates that you have reached the root).

Using Search Routines

`RegistryEntrySearch`, `RegistryEntryPropertyMod`, and `RegistryEntryMod` are used to search the Name Registry for entries having a specific property or set of modifiers. The set of entries to be searched is defined by a starting entry and a relationship. The relationship determines which entries relative to the starting entry are to be included in the search—children, parents, siblings, or descendants.

Follow these rules when using search routines:

- When an entry iterator is created via `RegistryEntryIterateCreate`, it is initialized to the default starting entry root and to the relationship of the `kRegIterDescendants` parameter. A subsequent search call using these default values will include all entries in the Name Registry.
- You can use `RegistryEntryIterateSet` to set the iterator to some name entry other than the root, limiting the iteration to some subset of the Name Registry. To change the default relationship, specify a new relationship as a parameter to your first search call.
- Search routines are designed to be iterative, allowing you to search for multiple instances of the same thing within a set of entries. To continue a search, make the same call again, specifying `kRegIterContinue` as the

relationship. The routine will continue where it left off and will find new entries that meet the same search criteria.

- To change the search criteria (property name, value, or modifiers) or the set of entries to be searched, reset the iterator. Use `RegistryEntryIterateSet` to set a new starting entry and then specify a new relationship in the next search call.
- A search operation is complete when either it finds what it is looking for or the `done` parameter returns `true`, indicating that there are no more name entries that meet the search criteria. When `done` is `true` no error code is returned and the contents of `foundEntry` are indeterminate. The iterator must be reset, using `RegistryEntryIterateSet`, before it can be used again for a subsequent search or iterate operation.

Here is a typical search sequence:

1. Get an iterator.
2. Set the starting point if it is other than the root.
3. Set the relationship in the first search call.
4. Do the search call.
5. Repeat the search call with the relationship set to `kRegIterContinue`.

Data Structures and Constants

Pathnames may be of any length, but components of a pathname are limited as follows:

```
enum
{
    kRegCStrMaxEntryNameLength              = 31
    kRegMaximumPropertyNameLength           = 31
};

typedef char          RegCStrPathName;

typedef unsigned long
                      RegPathNameSize;
```

```
typedef char         RegCStrEntryName,
                     *RegCStrEntryNamePtr
                     RegCStrEntryNameBuf[kRegCStrMaxEntryNameLength];

typedef char         RegPropertyName,
                     *RegPropertyNamePtr
                     RegPropertyNameBuf[kRegMaximumPropertyNameLength];

struct RegEntryID {
    UInt8       opaque[16];
};

typedef struct RegEntryID RegEntryID, *RegEntryIDPtr;
```

Software must use directed moves when examining a neighborhood in the Registry’s name tree. The following constants indicate the direction of movement during traversals of the hierarchical Registry tree:

```
typedef unsigned long
                     RegIterationOp;

typedef RegIterationOp
                     RegEntryIterationOp;

enum
{
    kRegIterRoot                = 0x2L, // absolute locations
    kRegIterParents             = 0x3L, // include all parent(s) of entry
    kRegIterChildren            = 0x4L, // include all children
    kRegIterDescendants         = 0x5L, // include all subtrees of entry
    kRegIterSibling             = 0x6L, // include all siblings
    kRegIterContinue            = 0x1L  // keep doing the same thing
};
```

ID Management

Mac OS provides several routines, described in this section, to create and manage name entry IDs. These IDs are discussed in “Name Entry Identifiers” (page 286).

RegistryEntryIDInit

`RegistryEntryIDInit` initializes a `RegEntryID` structure to a known, invalid state.

```
OSStatus RegistryEntryIDInit (RegEntryID *id);
```

`--> id`	Pointer to an identifier to be initialized.

DESCRIPTION

Since `RegEntryID` values are allocated on the stack, it is not possible to determine whether one contains a valid reference or uninitialized data from the stack. `RegistryEntryIDInit` corrects this problem. It should be called before a `RegEntryID` structure is used.

EXECUTION CONTEXT

`RegistryEntryIDInit` may be called from task level or secondary and hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

RegistryEntryIDCompare

`RegistryEntryIDCompare` compares `RegEntryID` values to see if they are equal. It can also be used to determine if a `RegEntryID` value is set to an invalid state.

```
Boolean RegistryEntryIDCompare(
                     const RegEntryID *id1,
                     const RegEntryID *id2);
```

`--> id1`	Pointer to the first identifier.
`--> id2`	Pointer to the second identifier.

DESCRIPTION

`RegistryEntryIDCompare` is useful for comparing two `RegEntryID` values to see whether they reference the same name entry as well as to check if a `RegEntryID` value is a valid reference. It returns `true` if the two ID values are equal.

If a `null` value is passed in either `id1` or `id2`, `RegistryEntryIDCompare` compares the other ID with the initialized value returned by `RegistryEntryIDInit`. If both ID values are `null`, `RegistryEntryIDCompare` returns `true`.

EXECUTION CONTEXT

`RegistryEntryIDCompare` may be called from any execution level.

RESULT CODES

`false`	0	ID values different
`true`	1	ID values equal

RegistryEntryIDCopy

`RegistryEntryIDCopy` copies the identifier for a name entry, including any internally allocated data.

```
void RegistryEntryIDCopy(
                    const RegEntryID *src,
                    RegEntryID *dst);
```

`--> src`	Pointer to ID to be copied.
`--> dst`	Pointer to the destination ID.

DESCRIPTION

Given an existing `RegEntryID` value, `RegistryEntryIDCopy` sets another `RegEntryID` to be functionally the same.

EXECUTION CONTEXT

`RegistryEntryIDCopy` may be called from any execution level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

RegistryEntryIDDispose

`RegistryEntryIDDispose` disposes of a Name Registry identifier.

```
void RegistryEntryIDDispose (RegEntryID *id);
```

`--> id`	Pointer to the `RegEntryID` value to be disposed of.

DESCRIPTION

`RegistryEntryIDDispose` disposes of the identifier for a name entry pointed to by `id`, including its allocated data.

EXECUTION CONTEXT

`RegistryEntryIDDispose` may be called from any execution level. If memory is disposed of it must be called at task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Name Creation and Deletion

The following routines add new name entries to the Name Registry and remove existing name entries from it.

RegistryCStrEntryCreate

`RegistryCStrEntryCreate` creates a new child name entry in the Name Registry.

```
OSStatus RegistryCStrEntryCreate(
                    const RegEntryID *parentEntry,
                    const RegCStrPathName *name,
                    RegEntryID *newEntry);
```

`--> parentEntry`	Pointer to `RegEntryID` value that identifies the parent name entry.
`--> name`	Pointer to the pathname of the new entry relative to the parent, as a C string.
`<-- newEntry`	Pointer to the returned `RegEntryID` value of the new name entry.

DESCRIPTION

Given the `RegEntryID` value of a parent name entry, `RegistryCStrEntryCreate` creates a new entry that is a descendant of the parent, with the C string pathname `name`. It returns the `RegEntryID` value that identifies the new name entry.

The rules for composing pathnames are given in "Pathnames" (page 286). Note that the pathname in `name` includes the name of the new entry. If `parentEntry` is `NULL`, `name` is a pathname relative to the root.

EXECUTION CONTEXT

`RegistryCStrEntryCreate` may be called only from task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrNotEnoughMemoryErr`	–2537	Not enough space in the system heap
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrPathNotFound`	–2539	Path component lookup failed
`nrNotCreatedErr`	–2540	Entry or property could not be created

CODE SAMPLE

Listing 10-2 shows code that uses `RegistryCStrEntryCreate` to add a name entry for a new child device to the Name Registry.

Listing 10-2 Adding a name entry to the Name Registry

```
OSStatus
AddDevice(
   const RegEntryID            *parentEntry,
   const RegCStrEntryName      *deviceName,
   RegEntryID                  *deviceEntry
   )


{
   RegCStrPathName     devicePathBuf[kRegCStrMaxEntryNameLength + 2]
                           = {kRegPathNameSeparator,kRegPathNameTerminator};
   RegCStrPathName     *devicePath = &devicePathBuf[0];
   OSStatus            err = noErr;

   /*
    * Need to construct a relative path name since we are not
    * attaching the new entry to the root.
    */
   devicePath = strcat(devicePath, deviceName);

   err = RegistryCStrEntryCreate(parentEntry, devicePath, deviceEntry);
   return err;
}
```

RegistryEntryDelete

`RegistryEntryDelete` deletes a name entry from the Name Registry.

```
OSStatus RegistryEntryDelete (const RegEntryID id);
```

`--> id`	Pointer to the `RegEntryID` value of name entry to delete.

DESCRIPTION

Given the `RegEntryID` value of a name entry in the Name Registry, `RegistryEntryDelete` deletes it. If the name entry node is deleted, the iterator and `RegEntryID` are no longer valid and cannot be used for subsequent iterations.

EXECUTION CONTEXT

`RegistryEntryDelete` may be called only from task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrLockedErr`	–2536	Entry or property is locked
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

Note
In the current Mac OS, all children of a parent entry are removed when the parent is removed. Removing a parent entry, thereby creating orphan entries, may not be supported in future releases. ◆

Name Iteration and Searching

The Name Registry name entry iteration functions communicate through an iterator parameter with the following type:

```
typedef struct      RegEntryIter { void *opaque; }
                     RegEntryIter, *RegEntryIterPtr;
```

RegistryEntryIterateCreate

`RegistryEntryIterateCreate` creates an iterator named `cookie` that is used by iterate and search routines. The iterator is initialized to the default starting entry `root` and to the relationship `kRegIterDescendants`, so it can be used to access the whole Name Registry.

```
OSStatus RegistryEntryIterateCreate (RegEntryIter *cookie);
```

`--> cookie`	Pointer to the iterator used by iterate and search routines.

DESCRIPTION

`RegistryEntryIterateCreate` sets up the iteration process for finding device names in the Name Registry and returns an iterator in `cookie` that is used by `RegistryEntryIterate` or `RegistryEntrySearch`.

EXECUTION CONTEXT

`RegistryEntryIterateCreate` may be called only from task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

RegistryEntryIterateSet

`RegistryEntryIterateSet` sets a `cookie` value to identify a specified starting name entry.

```
OSStatus RegistryEntryIterateSet(
                    RegEntryIter *cookie,
                    const RegEntryID *startEntryID);
```

`--> cookie`	Pointer to iterator used by iterate and search routines.

`-->` `startEntryID`
Pointer to the `RegEntryID` value that identifies the name entry to start iteration.

DESCRIPTION

When an iterator is first created, it is set to the root of the Name Registry with a relation of `kRegIterDescendants`. `RegistryEntryIterateSet` lets you adjust this starting point to a known name entry so you can iterate or search over a subset of the device tree.

The relation part of the iterator can be set by specifying a new relation in a subsequent iterate or search call.

EXECUTION CONTEXT

`RegistryEntryIterateSet` may be called only from task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

RegistryEntryIterate

One kind of iteration call, `RegistryEntryIterate`, retrieves the next name entry in the Name Registry by moving in a specified direction.

```
OSStatus RegistryEntryIterate(
                RegEntryIter *cookie,
                RegEntryIterationOp relationship,
                RegEntryID *foundEntry,
                Boolean *done);
```

`-->` `cookie` Pointer to the iterator used by iterate and search routines.

`-->` `relationship`
The iteration direction (values defined on (page 291)).

<-- `foundEntry`	Pointer to the ID of the next name entry found.
<-- `done`	Pointer to the interation result. A value of `true` means iteration is completed.

DESCRIPTION

`RegistryEntryIterate` moves from entry to entry in the Name Registry, marking its position by changing the value of `cookie`. The direction of movement is indicated by `relationship`. `RegistryEntryIterate` returns the `RegEntryID` value that identifies the next name entry found in `foundEntry`, or `true` in `done` if all name entries have been found.

EXECUTION CONTEXT

`RegistryEntryIterate` may be called from any execution level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

CODE SAMPLE

Listing 10-3 shows code using `RegistryEntryIterate` and `RegistryEntryDelete` that finds and removes all immediate child entries of a given parent entry. Deleting a name entry invalidates the iterator and `RegEntryID` for that name entry.

Listing 10-3 Finding and removing child entries

```
OSStatus
RemoveDevices(
   const RegEntryID        *parentEntry
   )
{
   RegEntryID              entry;
   RegEntryIter            cookie;
```

```
    RegEntryIterationOp     iterOp;
    Boolean                 done;
    OSStatus                err = noErr;

    RegistryEntryIDInit(&entry);

    err = RegistryEntryIterateCreate(&cookie);
    if (err != noErr)
        return err;

    /* Reset iterator to point to our parent entry */
    err = RegistryEntryIterateSet(&cookie, parentEntry);

    if (err == noErr)   {

        /* Include just immediate chidren, not all descendants */
        iterOp = kRegIterChildren;
        do  {
            err = RegistryEntryIterate(&cookie, iterOp, &entry, &done);
            if (!done && err == noErr)  {
                err = RegistryEntryDelete(&entry);
                RegistryEntryIDDispose(&entry);
            }
            iterOp = kRegIterChildren;

        } while (!done && err == noErr);
    }
    RegistryEntryIterateDispose(&cookie);
    return err;
}
```

RegistryEntrySearch

Another kind of iteration call, `RegistryEntrySearch`, retrieves the next name entry in the Name Registry that has a specified matching property.

```
OSStatus RegistryEntrySearch(
                     RegEntryIter *cookie,
                     RegEntryIterationOp relationship,
                     RegEntryID *foundEntry,
                     Boolean *done,
                     const RegPropertyName *propertyName,
                     const void *propertyValue,
                     RegPropertyValueSize propertySize);
```

`--> cookie` Pointer to the iterator used by iterate and search routines.

`--> relationship`
The search direction (values defined on (page 291)).

`<-- foundEntry`
Pointer to the ID of the next name entry found.

`<-- done` Pointer to the search result. A value of `true` means searching is completed.

`--> propertyName`
Pointer to name of property to be matched.

`--> propertyValue`
Pointer to value of property to be matched.

`--> propertySize`
Size of property to be matched.

DESCRIPTION

`RegistryEntrySearch` searches for a name entry with a property that matches certain criteria and returns the `RegEntryID` value that identifies that entry in `foundEntry`, or `true` in `done` if all matching name entries have been found.

`RegistryEntrySearch` returns only entries with properties that simultaneously match the values of `propertyName`, `propertyValue`, and `propertySize`. If the `propertyValue` pointer is `null` or `propertySize` is 0, then any property value is considered a match.

EXECUTION CONTEXT

`RegistryEntrySearch` may be called from any execution level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

CODE SAMPLE

Listing 10-4 shows code that uses `RegistryEntrySearch` to count the number of SCSI interface devices for a given parent device.

Listing 10-4 Using `RegistryEntrySearch`

```
OSStatus
FindSCSIDevices(
   const RegEntryID        *parentEntry,
   int                     *numberOfSCSIDevices
   )
{
   RegEntryIter            cookie;
   RegEntryID              SCSIEntry;
   RegEntryIterationOp     iterOp;
   Boolean                 done;
   OSStatus                err = noErr;

   #define kSCSIDeviceType     "scsi"

   RegistryEntryIDInit(&SCSIEntry);
   *numberOfSCSIDevices = 0;

   err = RegistryEntryIterateCreate(&cookie);
   if (err != noErr)
       return err;

   /*
    * Reset iterator to point to our parent entry
    */
   err = RegistryEntryIterateSet(&cookie, parentEntry);
```

```
    if (err == noErr)   {
        /*
         * Search all descendants of the parent device.
         */
        iterOp = kRegIterDescendants;
        do  {
            err = RegistryEntrySearch(&cookie, iterOp, &SCSIEntry, &done,
                    "device_type", kSCSIDeviceType, sizeof(kSCSIDeviceType));

            if (!done && err == noErr)  {
                *numberOfSCSIDevices += 1;
                RegistryEntryIDDispose(&SCSIEntry);
            }
            iterOp = kRegIterContinue;

        } while (!done && err == noErr);
    }
    RegistryEntryIterateDispose(&cookie);
    return err;
}
```

RegistryEntryIterateDispose

`RegistryEntryIterateDispose` disposes of the iteration structure after searching is finished.

```
void RegistryEntryIterateDispose (RegEntryIter *cookie);
```

`--> cookie`	Pointer to the iterator used by the iterate and search routines.

DESCRIPTION

Given the `cookie` value used previously, `RegistryEntryIterateDispose` disposes of resources used for iterating or searching.

EXECUTION CONTEXT

`RegistryEntryIterateDispose` may be called only from task level.

RESULT CODES

noErr	0	No error
paramErr	–50	Bad parameter

Name Lookup

RegistryCStrEntryLookup provides a fast, direct mechanism for finding a name entry in the Registry.

RegistryCStrEntryLookup

RegistryCStrEntryLookup finds a name entry in the Name Registry by starting from a designated point and traversing a defined path. This makes it faster than most search or iterate routines.

```
OSStatus RegistryCStrEntryLookup(
                    const RegEntryID *searchPointID,
                    const RegCStrPathName *pathName,
                    RegEntryID *foundEntry);
```

--> searchPointID
Pointer to the RegEntryID value that identifies starting point of search.

--> pathName Pointer to the pathname of entry to be found.

<-- foundEntry
Pointer to the RegEntryID value of found name entry.

DESCRIPTION

RegistryCStrEntryLookup finds a name entry in the Registry based on pathName, starting from the entry designated by searchPointID.

If searchPointID is NULL, the path is assumed to be a rooted path and pathName must contain an absolute pathname. If the pathname begins with a colon, the path is relative to searchPointID and pathName must contain a relative pathname. If the pathname does not begin with a colon, the path is a rooted path and pathName must contain an absolute pathname.

After using `RegistryCStrEntryLookup`, dispose of the `foundEntry` ID by calling `RegistryEntryIDDispose`.

Note
A reverse lookup mechanism has not been provided because some name services may not provide a fast, general algorithm. To perform reverse lookup, the process described in "Name Iteration and Searching" (page 297) should be used. ◆

EXECUTION CONTEXT

`RegistryCStrEntryLookup` may be called only from task level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrPathNotFound`	–2539	Path component lookup failed

CODE SAMPLE

Listing 10-5 shows code that uses `RegistryCStrEntryLookup` to obtain the entry ID for a child device.

Listing 10-5 Obtaining an entry ID

```
OSStatus
LocateChildDevice(
   const RegEntryID            *parentEntry,
   const RegCStrEntryName      *deviceName,
   RegEntryID                  *deviceEntry
   )
{
   RegCStrPathName     devicePathBuf[kRegCStrMaxEntryNameLength + 2]
                           = {kRegPathNameSeparator,kRegPathNameTerminator};
   RegCStrPathName     *devicePath = &devicePathBuf[0];
   OSStatus            err = noErr;
```

```
    /*
     * Need to construct a relative path name from the parent entry.
     */
    devicePath = strcat(devicePath, deviceName);

    err = RegistryCStrEntryLookup(parentEntry, devicePath, deviceEntry);
    return err;
}
```

Pathname Parsing

The routines defined in this section convert a `RegEntryID` value to the equivalent full pathname, give the pathname's length, and parse the pathname into its components.

RegistryEntryToPathSize

`RegistryEntryToPathSize` returns the size of the pathname to a specified name entry.

```
OSStatus RegistryEntryToPathSize(
                    const RegEntryID *entryID,
                    RegPathNameSize *pathSize);
```

`--> entryID`	Pointer to the `RegEntryID` value that identifies a name entry.
`<-- pathSize`	Pointer to the returned size in bytes of the pathname to the entry.

DESCRIPTION

`RegistryEntryToPathSize` returns in `pathSize` the length (in bytes) of the absolute pathname of the name entry designated by `entryID`, including the pathname's terminating character.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

RegistryCStrEntryToPath

`RegistryCStrEntryToPath` returns the pathname of a name entry in the Name Registry.

```
OSStatus RegistryCStrEntryToPath (
                    const RegEntryID *entryID,
                    RegCStrPathName *pathName,
                    RegPathNameSize pathSize);
```

`-->`	`entryID`	Pointer to the `RegEntryID` value that identifies a name entry.
`<--`	`pathName`	Pointer to the returned pathname to the entry.
`<--`	`pathSize`	The size in bytes of the pathname buffer pointed to by `pathName`.

DESCRIPTION

Given a `RegEntryID` value that identifies a name entry, `RegistryCStrEntryToPath` returns its pathname in `pathName`. If the buffer provided is too small, it returns `nrPathBufferTooSmall`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrPathBufferTooSmall`	–2543	Buffer for pathname too small

RegistryCStrEntryToName

`RegistryCStrEntryToName` retrieves the name component of a name entry and returns the ID of the entry's parent.

```
OSStatus RegistryCStrEntryToName (
                    const RegEntryID *entryID,
                    RegEntryID *parentEntry,
                    RegCStrEntryName *nameComponent,
                    Boolean *done);
```

`--> entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`<-- parentEntry`
Pointer to the returned `RegEntryID` value of the entry's parent entry.

`<-- nameComponent`
Pointer to the returned name of the entry as a C string.

`<-- done` Pointer, returns `true` when `parentEntry` is the root.

DESCRIPTION

Given a `RegEntryID` value that identifies a name entry, `RegistryCStrEntryToName` returns the `RegEntryID` value that identifies its parent entry in `parentEntry` and the name component of the name entry in `nameComponent`. `RegistryCStrEntryToName` is useful for locating the parent of a name entry and for constructing a relative pathname from the parent to the entry.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

CODE SAMPLE

Listing 10-6 shows code that uses `RegistryCStrEntryToName` to obtain the parent entry for a given child entry.

Listing 10-6 Obtaining a parent entry

```
OSStatus
LocateParentDevice(
   const RegEntryID        *deviceEntry,
   RegEntryID              *parentEntry
   )
{
   RegCStrEntryName        deviceNameBuf[kRegCStrMaxEntryNameLength+1];
   Boolean                 done;
   OSStatus                err = noErr;

   err = RegistryCStrEntryToName(deviceEntry, parentEntry,
                                 &deviceNameBuf[0], &done);
   if (err != noErr)
       return err;

   /*
    * If done == true, we have reached the root, there is no parent!
    */
   if (done)
       err = kNotFoundErr;

   return err;
}
```

Property Management

Properties describe what a name entry represents or how it may be used. Each name entry has a set of named properties, which may be empty. Each property consists of a name string and a value. The value consists of 0 or more contiguous bytes. Property names are null-terminated strings of at most `kRegMaximumPropertyNameLength` bytes (31 bytes). Name property data structures and constants are listed in "Data Structures and Constants" (page 290).

Creation and Deletion

The routines described in this section add new properties to or remove existing properties from a name entry in the Name Registry.

RegistryPropertyCreate

`RegistryPropertyCreate` adds a new property to a name entry.

```
OSStatus RegistryPropertyCreate (
                    const RegEntryID *entryID,
                    const RegPropertyName *propertyName,
                    const void *propertyValue,
                    RegPropertyValueSize propertySize);
```

`-->` `entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`-->` `propertyName`
Pointer to the name of the property to be created.

`-->` `propertyValue`
Pointer to the value to create for the new property.

`-->` `propertySize`
The size in bytes of the new property.

DESCRIPTION

Given a `RegEntryID` value that identifies a name entry, `RegistryPropertyCreate` adds a new property to that entry with name `propertyName` and value `propertyValue`. The `entryID` parameter may not be `null`.

The `propertySize` parameter must be set to the size (in bytes) of `propertyValue`.

Property names may be any alphanumeric strings but may not contain slash (/) or colon (:) characters.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrNotEnoughMemoryErr`	–2537	Not enough space in the system heap
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotCreatedErr`	–2540	Entry or property could not be created
`nrNameErr`	–2541	Name invalid, too long, or not terminated

CODE SAMPLE

In Listing 10-7, `RegistryPropertyCreate`, `RegistryPropertyGetSize`,and `RegistryPropertySet` are used to update the value of a given property of a name entry. If the property exists, its value is updated. If it doesn't exist, a new property is created.

Listing 10-7 Updating or creating a property

```
OSStatus
UpdateDeviceProperty(
   const RegEntryID            *deviceEntry,
   const RegPropertyName       *propertyName,
   const void                  *newPropertyValue,
   const RegPropertyValueSize  newPropertySize
   )
{
   RegPropertyValueSize    PrevPropertySize;
   OSStatus                err = noErr;

   /*
    * RegistryPropertyGetSize used here to see if the property exists.
    */
   err = RegistryPropertyGetSize(deviceEntry,propertyName,&PrevPropertySize);
   if (err == noErr) {
       err = RegistryPropertySet(deviceEntry, propertyName,
               newPropertyValue, newPropertySize);
       return err;

   } else if (err == nrNotFoundErr)
       err = RegistryPropertyCreate(deviceEntry, propertyName,
               newPropertyValue, newPropertySize);

   return err;
}
```

RegistryPropertyDelete

`RegistryPropertyDelete` deletes a property from the Name Registry.

```
OSStatus RegistryPropertyDelete (
                     const RegEntryID *entryID,
                     const RegPropertyName *propertyName);
```

`-->` `entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`-->` `propertyName`
Pointer to the name of the property to be deleted.

DESCRIPTION

`RegistryPropertyDelete` deletes the property named `propertyName` from the name entry identified by `entryID`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrLockedErr`	–2536	Entry or property locked
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria

Property Iteration

Traversing the set of properties associated with a name entry is similar to iteration over names in the Registry, described in "Name Iteration and Searching" (page 297).

Only one form of property iteration is provided—iteration over the set of properties associated with a single name entry.

A property iteration loop has this general form:

```
Create(...)
do {
        Iterate(...);
} while (!done);
Dispose(...);
```

Property iteration functions communicate by means of an iterator parameter that is a `RegPropertyIter` data structure:

```
typedef struct      RegPropertyIter { void *opaque; }
                    RegPropertyIter,
                    *RegPropertyIterPtr;
```

RegistryPropertyIterateCreate

The starting routine `RegistryPropertyIterateCreate` creates an iterator for all the properties associated with a name entry.

```
OSStatus RegistryPropertyIterateCreate (
                    const RegEntryID *entry,
                    RegPropertyIter *cookie);
```

`--> entry`	Pointer to the `RegEntryID` value that identifies a Name Registry name entry.
`<-- cookie`	Pointer to the iterator used by property iterate routines.

DESCRIPTION

`RegistryPropertyIterateCreate` creates a property iterator (`cookie`) that can be used to iterate the properties of the name entry identified by `entry`. The value it returns in `cookie` is used by `RegistryPropertyIterate`, described next.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

RegistryPropertyIterate

Repeated calls to `RegistryPropertyIterate` use the iterator returned by `RegistryPropertyIterateCreate` to iterate through a succession of properties.

```
OSStatus RegistryPropertyIterate (
                    RegPropertyIter *cookie,
                    RegPropertyName *foundProperty,
                    Boolean *done);
```

`--> cookie`	Pointer to the iterator used by property iterate routines.
`<-- foundProperty`	Pointer to the name of the property found.
`<-- done`	Pointer, returns `true` when all properties have been found.

DESCRIPTION

`RegistryPropertyIterate` moves from property to property among the properties of the name entry specified in a prior `RegistryPropertyIterateCreate` call (see previous section). It returns the name of the next property in `foundProperty`, or `true` in `done` if all properties have been iterated through.

RESULT CODES

noErr	0	No error
paramErr	–50	Bad parameter

CODE SAMPLE

Listing 10-8 shows code that uses `RegistryPropertyIterate` to iterate through all the properties for a given name entry.

Listing 10-8 Iterating through properties

```
OSStatus
IterateDeviceProperties(
   const RegEntryID        *deviceEntry
   )
{
   RegPropertyNameBuf      propertyName;
   RegPropertyIter         cookie;
   Boolean                 done;
   OSStatus                err = noErr;

   err = RegistryPropertyIterateCreate(deviceEntry, &cookie);

   if (err != noErr)   {
       do  {
           err = RegistryPropertyIterate(&cookie, &propertyName[0], &done);
           if (err != noErr)
               break;

           /*
            * Do something with the property, given the property name
            * you can use RegistryPropertyGetSize to determine the size
            * of the value and and RegistryPropertyGet to retrieve the value.
            */
```

```
        } while (!done && err == noErr);
    }
    RegistryPropertyIterateDispose(&cookie);
    return err;
}
```

RegistryPropertyIterateDispose

`RegistryPropertyIterateDispose` completes the property iteration process.

```
void RegistryPropertyIterateDispose (RegPropertyIter *cookie);
```

`--> cookie`	Pointer to the iterator used by iterate and search routines.

DESCRIPTION

`RegistryPropertyIterateDispose` disposes of the iterator used to find properties. It should be called even in the case of an error, so that allocated data structures can be freed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Property Retrieval and Assignment

The value of an existing property may be retrieved or modified using the routines defined in this section.

RegistryPropertyGetSize

A property's value may have any length. If the length of a property's value is not known, use `RegistryPropertyGetSize` to determine its size so you can allocate space for it.

```
OSStatus RegistryPropertyGetSize (
                     const RegEntryID *entryID,
                     const RegPropertyName *propertyName,
                     RegPropertyValueSize *propertySize);
```

`--> entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`--> propertyName`
Pointer to the name of the property.

`<-- propertySize`
Pointer to the size in bytes returned for the property's value.

DESCRIPTION

`RegistryPropertyGetSize` returns in `propertySize` the length (in bytes) of the property named `propertyName` and associated with the name entry identified by `entryID`.

EXECUTION CONTEXT

`RegistryPropertyGetSize` may be called from task level or secondary interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria

CODE SAMPLE

In Listing 10-9, `RegistryPropertyGetSize` and `RegistryPropertyGet` are used to obtain the value of a property.

Listing 10-9 Obtaining a property value

```
OSStatus
GetDeviceProperty(
   const RegEntryID        *deviceEntry,
   const RegPropertyName   *propertyName,
   RegPropertyValue        propertyValue,
   RegPropertyValueSize    *propertySize
   )
{
   RegPropertyValueSize    size;
   OSStatus                err = noErr;

   /*
    * Get the size of the value first to see if our buffer is big enough.
    */
   err = RegistryPropertyGetSize(deviceEntry, propertyName, &size);
   if (err == noErr) {
       if (size > *propertySize)
           return kPropBufferTooSmall;
       /*
        * Note, we return the actual property size.
        */
       err = RegistryPropertyGet(deviceEntry, propertyName, propertyValue,
                   propertySize);
   }
   return err;
}
```

RegistryPropertyGet

`RegistryPropertyGet` retrieves the value of a property in the Name Registry.

```
OSStatus RegistryPropertyGet (
                     const RegEntryID *entryID,
                     const RegPropertyName *propertyName,
                     void *propertyValue,
                     RegPropertyValueSize *propertySize);
```

`-->` `entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`-->` `propertyName`
Pointer to the name of the property.

`<--` `propertyValue`
Pointer containing the returned value for the property.

`<-->` `propertySize`
On input a pointer to the size in bytes of the property buffer. Upon return a pointer to the actual size in bytes of the property's value.

DESCRIPTION

`RegistryPropertyGet` retrieves the value of the property named `propertyName` and associated with the name entry identified by `entryID`. The `propertySize` parameter must be set to the size in bytes of the buffer pointed to by `propertyValue`. Upon return, the value of `propertySize` will be the actual length of the value in bytes.

EXECUTION CONTEXT

`RegistryPropertyGet` may be called from task level or outside the task level context.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria

RegistryPropertySet

`RegistryPropertySet` sets the value of a property in the Name Registry.

```
OSStatus RegistryPropertySet (
                    const RegEntryID *entryID,
                    const RegPropertyName *propertyName,
                    const void *propertyValue,
                    RegPropertyValueSize propertySize);
```

`--> entryID` Pointer to the `RegEntryID` value that identifies a name entry.

`--> propertyName`
Pointer to the name of the property. For computers built prior to the iMac, this value cannot exceed 4 bytes. For the iMac and later Macintosh models, this value cannot exceed 8 bytes.

`--> propertyValue`
Pointer to the value to which to set the property.

`--> propertySize`
Pointer to the size in bytes of the property. For computers built prior to the iMac, this value cannot exceed 8 bytes. For the iMac and later Macintosh models, this value cannot exceed 32 bytes.

DESCRIPTION

`RegistryPropertySet` sets the value of the property named `propertyName` and associated with the name entry identified by `entryID`. The `propertySize` parameter must be set to the size (in bytes) of the value pointed to by `propertyValue`.

EXECUTION CONTEXT

`RegistryPropertySet` may be called from task level only, not from secondary interrupt level or hardware interrupt level. This restriction is due to the fact that a call to `RegistryPropertySet` allocates memory in NVRAM.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrLockedErr`	–2536	Entry or property locked
`nrNotEnoughMemoryErr`	–2537	Not enough space in the system heap
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria
`nrNameErr`	–2541	Name invalid, too long, or not terminated
`nrOverrunErr`		This error is only possible if the property has the `kRegPropertyValueIsSavedToNVRAM` modifier
`nrTypeMismatchErr`		This error is only possible if the property has the `kRegPropertyValueIsSavedToNVRAM` modifier

Standard Properties

Some standard Name Registry properties names are specified for device entries. These names should not be used for other purposes. Standard reserved property names used by PCI expansion cards are listed in Table 10-1. Property names beginning with `APPL` are reserved for Apple use.

Table 10-1 Reserved Name Registry property names

Name	Description
Open Firmware standard properties	
`address`	Defines large virtual address regions
`compatible`	Defines alternate `name` property values*
`device_type`	The implemented interface
`fcode-rom-offset`	Location of node's FCode in the expansion ROM
`interrupts`	Defines the interrupts used
`model`	Defines a manufacturer's model
`name`	Name of the name entry (`nameString`); see (page 252)
`reg`	The package's physical address space request
`status`	Indicates the device's operations status

Table 10-1 Reserved Name Registry property names (continued)

Name	**Description**
Properties defined by PCI binding to Open Firmware	
`alternate-reg`	Alternate access paths for addressable regions
`assigned-addresses`	Assigned physical addresses
`class-code`	Value from corresponding PCI configuration register
`device-id`	Value from corresponding PCI configuration register
`devsel-speed`	Value from corresponding PCI configuration register
`driver,xxx,yyy,zzz`	Driver code for *xxx,yyy,zzz* platform
`driver-reg,xxx,yyy,zzz`	Descriptor of location for driver code for *xxx,yyy,zzz* platform (not supported by Mac OS)
`fast-back-to-back`	Value from corresponding PCI configuration register
`max-latency`	Value from corresponding PCI configuration register
`min-grant`	Value from corresponding PCI configuration register
`power-consumption`	Function's power requirements
`revision-id`	Value from corresponding PCI configuration register
`vendor-id`	Value from corresponding PCI configuration register
Properties specific to the Power Macintosh platform	
`AAPL,address`	Vector to logical address pointers†
`AAPL,interrupts`	Internal interrupt number
`AAPL,slot-name`	Physical slot identifier
`depth`	Color depth of each pixel (for display device node only)
`driver,AAPL,MacOS,PowerPC`	Driver code for Mac OS
`driver-descriptor`	Property that contains the driver description structure
`driver-ist`	IST member and set value, used to install interrupts‡
`driver-ptr`	Memory address of driver code
`driver-ref`	Reference to driver controlling a specific name entry

Table 10-1 Reserved Name Registry property names (continued)

Name	**Description**
`height`	Height in pixels (for display device node only)
`linebytes`	Number of bytes in each line (for display device node only)
`width`	Width in pixels (for display device node only)

* See "Matching Drivers With Devices" (page 164).
† See "Fast I/O Space Cycle Generation" (page 454).
‡ See "Interrupts and the Name Registry" (page 391).

Normally, the device tree shows several properties attached to each device. Most of these properties are created and used by Open Firmware and are described fully in IEEE Standard 1275, described on (page 27). Some properties are Apple specific and are required only by Power Macintosh computers or Mac OS. Following are some notes on the properties listed in Table 10-1:

- Manufacturers of PCI cards should use their United States stock symbol (if they are a publicly traded company) as the hardware manufacturer's ID in the `name` property. Otherwise, they can ask the IEEE to assign a 24-bit ID number by contacting

Registration Authority Committee
IEEE, Inc.
445 Hoes Lane
Piscataway, NJ 08855-1331
Telephone 809-562-3812

- Mac OS native drivers should use the following value for their `driver` property:

```
driver,AAPL,MacOS,PowerPC
```

- A standard property that is important to native drivers is the `assigned-addresses` property defined in *PCI Bus Binding to IEEE 1275-1994,* currently available from the IEEE as described in a note on (page 27). The `assigned-addresses` property tells the driver where a card's relocatable address locations have been placed in physical memory. With all routines except the Expansion Bus Manager I/O functions, driver code must resolve `assigned-addresses` values to `AAPL,address` values before using them. Sample code that retrieves an `assigned-addresses` property from the Name Registry is shown in Listing 9-3 (page 274).

- Drivers can use the `vendor-id`, `device-id`, `class-code`, and `revision-id` properties to distinguish one card from another. However, these values typically refer to the controller on the card rather than the card itself. For example, software will be unable to use these properties to distinguish between two video cards that use the same controller chip. Driver writers can make the cards distinct by giving different names to them in their FCode assignments.
- The `fcode-rom-offset` property contains the location in the PCI card's expansion ROM at which the FCode that produces the node is found. The FCode tokenizer tool can use the value of this property to determine the values of other properties, such as `driver`. If a card's expansion ROM contains no FCode, the `fcode-rom-offset` property will be absent from the card's Name Registry entry.
- The `driver-ref` pointer can be important. This property is created by the system when a device driver is installed; it is the driver reference as defined by *Inside Macintosh: Devices.* The property is removed when the driver is removed. The presence of this property can be used to determine whether a particular device is open.
- The `driver-descriptor` property is a structure taken from the driver header; it defines various characteristics of the device. The contents of this property are defined in "Driver Description Structure" (page 198).
- The `AAPL,address` property is a vector to an array of logical address pointers, as described in "Fast I/O Space Cycle Generation" (page 454).
- The `AAPL,interrupts` property is an internal interrupt number that the Open Firmware startup process creates before any FCode is read from the card. This property is implementation dependent and does not necessarily reflect the actual interrupt bit number in the interrupt controller.
- The `AAPL,slot-name` property is an identifier for the hardware slot in which the card is plugged. This property is created by the Open Firmware startup process before any FCode is read from the card. Its value may be different with different Power Macintosh models.
- The `height`, `width`, `linebytes`, and `depth` properties are attached to the Name Registry entries of graphic display devices to define each display's characteristics.
- The property `driver,`*xxx*`,`*yyy*`,`*zzz* provides access to driver code. An expansion card ROM may contain a number of different drivers suited to different operating systems and machine architectures. The value of *xxx*

specifies the manufacturer of the hardware (`AAPL` for Apple Computer, Inc.), *yyy* specifies the operating system (`MacOS`), and *zzz* specifies the instruction set architecture (`PowerPC`). The value of `driver,`*xxx*`,`*yyy*`,`*zzz* is the driver code itself, which can be quite large; there is no defined upper limit to the size of a property's data. Although a PCI card may define a number of drivers, only drivers appropriate to an available operating system will be placed in the device tree, and therefore only these drivers can be accessed through the Name Registry.

Modifier Management

Modifiers, described in this section, convey special characteristics of names and properties. They are provided for use by low-level experts designed for specific platforms. Modifiers may be supported for some names and not others. Support may change from one hardware platform to another. Hence, device drivers should not rely on modifiers to determine device functionality.

Data Structures and Constants

Modifiers are specified as bits in a 32-bit word. The low-order 16 bits are reserved for modifiers applicable to both names and properties. The next 8 bits are reserved by the name space and are redefined for each name space. The high-order 8 bits are reserved for each name and property set and are redefined for each name entry.

The following types are used to declare modifier words:

```
typedef unsigned long   RegModifiers;

typedef RegModifiers    RegEntryModifiers;

typedef RegModifiers    RegPropertyModifiers;
```

The following constants are used to mask bits in modifier words:

Name	Value	Description
`kRegNoModifiers`	0x00000000	No entry modifiers in place

Name	Value	Description
`kRegUniversalModifierMask`	0x0000FFFF	Modifiers to all entries
`kRegNameSpaceModifierMask`	0x00FF0000	Modifiers to all entries within the name space
`kRegModifierMask`	0xFF000000	Modifiers to just this entry

The following constants have meaning for property modifiers:

Name	Value	Description
`kRegPropertyValueIsSavedToNVRAM`	0x00000001	Saved in NVRAM
`kRegPropertyValueIsSavedToDisk`	0x00000002	Saved to disk

Modifier-Based Searching

Mac OS provides two routines to simplify searching for name entries or properties that have particular modifiers.

RegistryEntryMod

`RegistryEntryMod` searches for name entries that have specified modifiers.

```
OSStatus RegistryEntryMod (
                    RegEntryIter *cookie,
                    RegEntryIterationOp relationship,
                    RegEntryID *foundEntry,
                    Boolean *done,
                    RegEntryModifiers matchingModifiers);
```

`--> cookie` Pointer to the iterator returned by the `RegistryIterateCreate` function for the name entry iterate and search routines.

`--> relationship`
The search relationship (values defined on (page 291)).

`<-- foundEntry`
Pointer to the ID returned for the next name entry found.

`<-- done` Pointer, contains `true` if searching is completed.

`<-- matchingModifiers`
The modifiers to be matched.

DESCRIPTION

`RegistryEntryMod` searches for name entries, using the relation indicated by `relationship`, that have a specified modifier. `RegistryEntryMod` returns the `RegEntryID` value that identifies the next name entry found in `foundEntry`, or `true` in `done` if all entries have been exhausted.

`RegistryEntryMod` returns only name entries with modifiers that match the value of `matchingModifiers`. It uses a bit AND operation to determine when the bits set in `matchingModifiers` are also set in the entry.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

RegistryEntryPropertyMod

`RegistryEntryPropertyMod` searches for name entries that have a property with a specified modifier.

```
OSStatus RegistryEntryPropertyMod (
                        RegEntryIter *cookie,
                        RegEntryIterationOp relationship,
                        RegEntryID *foundEntry,
                        Boolean *done,
                        RegEntryModifiers matchingModifiers);
```

`--> cookie` Pointer to the iterator returned by the `RegistryIterateCreate` function for the name entry iterate and search routines.

`--> relationship`
The search relationship (values defined on (page 291)).

`<-- foundEntry`
Pointer to the ID of the next name entry found.

`<-- done`	Pointer, contains a value of `true` when searching is completed.
`--> matchingModifiers`	Pointer to the modifiers to be matched.

DESCRIPTION

`RegistryEntryPropertyMod` searches for name entries, using the relation indicated by `relationship`, that have a property with a specified modifier. It returns the `RegEntryID` value that identifies the next name entry found in `foundEntry`, or `true` in `done` if all entries have been exhausted.

`RegistryEntryPropertyMod` returns only name entries with properties that have modifiers that match the value of `matchingModifiers`. It uses a bit AND operation to determine when the bits set in `matchingModifiers` are also set in the property.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Name Modifier Retrieval and Assignment

Existing name entries and properties may have their modifier word's value set or retrieved. Code can accomplish this by using the routines described in this section.

IMPORTANT

In the current implementation of the Name Registry, the only modifiers that you can change are `kRegPropertyValueIsSavedToNVRAM` and `kRegPropertyValueIsSavedToDisk`. Changing other modifiers is reserved for future versions of Mac OS. ▲

RegistryEntryGetMod

`RegistryEntryGetMod` fetches the modifiers for a name entry in the Registry.

```
OSStatus RegistryEntryGetMod (
                    const RegEntryID *entry,
                    RegEntryModifiers *modifiers);
```

`--> entry`	Pointer to the `RegEntryID` value that identifies a name entry.
`<-- modifiers`	Upon return contains the value of modifiers for the specified entry.

DESCRIPTION

`RegistryEntryGetMod` returns in `modifiers` the current modifiers for the name entry identified by `entry`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

CODE SAMPLE

In Listing 10-10, `RegistryEntryGetMod` and `RegistrySetMod` are used to save a property to disk.

Listing 10-10 Saving a property to disk

```
OSStatus
SaveDeviceProperty(
   const RegEntryID        *deviceEntry,
   const RegPropertyName   *propertyName
   )
{
   RegPropertyModifiers    propertyModifiers;
   OSStatus                err = noErr;
```

```
  /*
   * Get the existing modifiers first.
   */
 err = RegistryPropertyGetMod (deviceEntry,propertyName,&propertyModifiers);

                if (err == noErr)   {
                    /*
                     * Set the save-to-disk modifier preserving the
                     * already existing ones.
                     */
                    propertyModifiers = propertyModifiers
                                        & kRegPropertyValueIsSavedToDisk;
                    err = RegistryPropertySetMod
                          (deviceEntry, propertyName, propertyModifiers);
                }
                return err;
            }
```

RegistryEntrySetMod

`RegistryEntrySetMod` sets the modifiers for a name entry in the Registry.

```
OSStatus RegistryEntrySetMod (
                     const RegEntryID *entry,
                     const RegEntryModifiers modifiers);
```

`--> entry`	Pointer to the `RegEntryID` value that identifies a name entry.
`<-- modifiers`	Pointer to the value of modifiers to set.

DESCRIPTION

`RegistryEntrySetMod` sets the modifiers specified in `modifiers` for the name entry identified by `entry`. The caller is responsible for preserving bits that should remain set by reading the current modifier value, changing it, and then assigning the new value.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid

Property Modifier Retrieval and Assignment

The two routines described in this section retrieve and assign property modifiers.

RegistryPropertyGetMod

`RegistryPropertyGetMod` fetches the modifiers for a property in the Registry.

```
OSStatus RegistryPropertyGetMod (
                    const RegEntryID *entry,
                    const RegPropertyName *name,
                    RegPropertyModifiers *modifiers);
```

`--> entry`	Pointer to the `RegEntryID` value that identifies a name entry.
`--> name`	Pointer to the property name.
`<-- modifiers`	Pointer to the value returned for the property modifiers.

DESCRIPTION

`RegistryPropertyGetMod` returns in `modifiers` the current modifiers for the property with name `name` in the name entry identified by `entry`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria

RegistryPropertySetMod

`RegistryPropertySetMod` sets the modifiers for a property in the Registry.

```
OSStatus RegistryPropertySetMod (
                    const RegEntryID *entry,
                    const RegPropertyName *name,
                    RegPropertyModifiers modifiers);
```

`-->` `entry`	Pointer to the `RegEntryID` value that identifies a name entry.
`-->` `name`	Pointer to the property name.
`-->` `modifiers`	Pointer to the value of property modifiers to set.

DESCRIPTION

`RegistryPropertySetMod` sets the modifiers specified in `modifiers` for the property with name `name` in the name entry identified by `entry`. The caller is responsible for preserving bits that should remain set by reading the current modifier value, changing it, and then assigning the new value.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`nrInvalidNodeErr`	–2538	`RegEntryID` value not valid
`nrNotFoundErr`	–2539	Search failed to match criteria

Macintosh System Gestalt

When it builds the device tree, the Macintosh ROM installs a node at its root, called the **gestalt node,** that contains information about the Macintosh system on which it is running. The names of the properties of this node are the standard Macintosh gestalt selectors, as described in *Inside Macintosh: Operating System Utilities.* This book is described in "Supplementary Documents" (page 26). Some of the available Gestalt properties of interest to PCI drivers are shown in Table 10-2.

Table 10-2 Gestalt properties

Name	Description
`"fpu "`	Floating-point unit type
`"hdwr"`	Low-level hardware configuration attributes
`"kbd "`	Keyboard type
`"lram"`	Logical RAM size
`"mach"`	Macintosh model code
`"mmu "`	Memory management unit type
`"nreg"`	Name Registry version
`"pgsz"`	Logical page size
`"proc"`	Microprocessor type
`"prty"`	Parity attributes
`"ram "`	Physical RAM size
`"rom "`	System ROM size

Table 10-2 Gestalt properties (continued)

Name	Description
`"romv"`	System ROM version
`"ser "`	Serial hardware attributes
`"snd "`	Sound attributes
`"tv  "`	TV support version
`"vers"`	Gestalt version
`"vm  "`	Virtual memory attributes

Note
Specific Macintosh computer models may lack some of the gestalt values listed in Table 10-2, so the corresponding properties will not appear in the gestalt node. Macintosh computers introduced after the iMac do not include the gestalt node. ◆

PCI expansion card firmware and driver code can explore the gestalt name entry in the Name Registry to determine the hardware and firmware environment available to it. For example, Listing 10-11 shows typical code to extract the 32-bit value of the Macintosh virtual memory attributes from the `"vm  "` property of the gestalt name entry.

Listing 10-11 Sample code to fetch virtual memory gestalt

```
RegEntryIter          cookie;
RegEntryID            gestaltEntry;
RegPropertyValueSize        gestaltEntrySize = sizeof(UInt32);
Boolean               done;
OSErr                 err;

    err = RegistryEntryIterateCreate(&cookie);
if ( err != noErr )
    return err;
```

```
err = RegistryEntrySearch (          &cookie,
                                     kRegIterRoot,
                                     &gestaltEntry,
                                     &done,
                                     "vm  ",
                                     nil,
                                     0 );
if ( err != noErr )
    return err;

err = RegistryPropertyGet ( &gestaltEntry,
                                     "vm  ",
                                     &vmIsOn,
                                     &gestaltEntrySize );

if ( err != noErr )
    return err;

RegistryEntryIterateDispose (&cookie);
```

Code Samples

This section contains code samples that illustrate common Name Registry operations.

Adding a Device Entry

For all physical devices, adding a device entry to the Name Registry is handled by the device's expert. Device drivers normally do not need to add their devices to the Registry.

Adding a new device to the system consists of entering a new name entry in the Registry and setting the appropriate property values. The example shown in Listing 10-12 adds a new name entry to the Registry with a single property.

Listing 10-12 Adding a name entry to the Name Registry

```
#include <NameRegistry.h>

OSStatus JoePro_AddName(
    const RegCStrPathName              *name,
    const RegPropertyName              *prop,
    const void                         *val,
    const RegPropertyValueSize         len
    )
{
    OSStatus            err = noErr;
    RegEntryID          where, new_entry;

    err = JoePro_FigureOutWhere(&where);
    if (err == noErr) {
        err = JoePro_EnterName(&where, name, &new_entry);
        RegistryEntryIDDispose(&where);
    }
    if (err == noErr) {
        err = JoePro_AddProperties(&new_entry, prop, val, len);
        RegistryEntryIDDispose(&new_entry);
    }
    return err;
}

OSStatus
JoePro_FigureOutWhere(RegEntryID *where)
{
    OSErr               err = noErr;
    RegEntryIter        cookie;
    Boolean             done = FALSE;

    /*
     * We want to search all the names, which is
     * the default, so we just need to continue.
     */
    RegEntryIterationOp op = kRegIterContinue;
    /*
     * For this example, the existence of the
     * “Joe Pro Root” property is used to find
     * out where to put the “Joe Pro” devices.
     * Initialization code will need to have
     * created this entry.
     */
    RegPropertyNameBuf              name;
    RegPropertyValue                val = NULL;
    RegPropertyValueSize            siz = 0;
    strncpy(name, "Joe Pro Root", sizeof(name));

    /*
     * Figure out where to put the driver.
     *

     * By convention, there is one “Joe Pro Root”
     * so we don’t need to loop.
     */
    err = RegistryEntryIterateCreate(&cookie);
    if (err == noErr) {
        err = RegistyEntrySearch(&cookie, op, &where, &done,
                    name, val, siz);
    }
    RegistryEntryIterateDispose(&cookie);

    /*
     * Check if we completed the search without
     * finding the “Joe Pro Root”.
     */
    assert(err != noErr || !done);
    return err;
}

OSStatus JoePro_EnterName(
    const RegEntryID                *where,
    const RegCStrPathName           *name,
    RegEntryID                      *entry
    )
{
    /*
```

```
     * Assumption: This call will return an error
     *      if the name entry is already in the Registry.
     */
    return RegistryCStrEntryCreate(where, name, entry);
}

OSStatus
JoePro_AddProperties(
    const RegEntryID                    *entry,
    const RegPropertyName               *prop,
    const void                          *val,
    const RegPropertyValueSize          siz
    )
{
    return RegistryPropertyCreate(entry, prop, val, siz);
}
```

Since all name entries in the registry are connected to at least one other entry, either an existing name entry must be provided when creating a new entry or it will be assumed that the path is specified relative to the root entry.

The creator of a name entry must determine where in the tree it should appear. This is typically determined by convention.

Finding a Device Entry

Every device driver typically needs to retrieve information about the device from the Name Registry. The example in Listing 10-13 retrieves the value of a single property for a specified name entry in the Name Registry.

Listing 10-13 Retrieving the value of a property

```
#include <NameRegistry.h>

OSStatus
JoePro_LookupProperty(
    const RegCStrPathName           *name,
    const RegPropertyName           *prop,
    RegPropertyValue                *val,
    RegPropertyValueSize            *siz
    )
{
    OSErr err = noErr;
    RegEntryID entry;

    err = JoePro_FindEntry(name, &entry);
    if (err == noErr) {
        err = JoePro_GetProperty(&entry, prop, val, siz);
        RegistryEntryIDDispose(&entry);
    }
    return err;
}

OSStatus JoePro_FindEntry(
    const RegCStrPathName           *name,
    RegEntryID                      *entry
    )
{
    return RegistryCStrEntryLookup(
            NULL /* start root */, name, entry);
}

OSStatus JoePro_GetProperty(
    RegEntryID                  *entry,
    RegPropertyName             *prop,
    RegPropertyValue            *val,
    RegPropertyValueSize        *siz
    )
{

    OSErr err = noErr;

    /*
     * Figure out how big a buffer we need for the value
     */
    err = RegistryPropertyGetSize(entry, prop, siz);
    if (err == noErr) {
        *val = (RegPropertyValue) malloc(*siz);

        assert(*val != NULL);
        err = RegistryPropertyGet(entry, prop, val, siz);
        if (err != noErr) {
            free(*val);
            *val = NULL;
        }
    }
    return err;
}
```

Removing a Device Entry

When a device is permanently removed from the system, the information pertaining to the device must be removed from the Name Registry. When a name entry is removed from the Registry, all properties associated with that entry are automatically removed as well. Listing 10-14 illustrates removing a device entry from the Registry.

Note
In the current Mac OS, all children of a parent entry are removed when the parent is removed. Removing a parent entry, thereby creating orphan entries, may not be supported in future releases. ◆

Listing 10-14 Removing a device entry from the Name Registry

```
#include <NameRegistry.h>

OSStatus
JoePro_RemName(const RegCStrPathName *name)
{
    OSErr err = noErr;
    RegEntryID entry;

/* from previous example */
    err = JoePro_FindEntry(name, &entry);
    if (err == noErr) {
        err = JoePro_RemEntry(&entry);
        RegistryEntryIDDispose(&entry);
    }
    return err;
}

OSStatus
JoePro_RemEntry(RegEntryID *entry)
{
    return RegistryEntryDelete(entry);
}
```

Listing Devices

Expert software must be able to find various devices in the system. The example shown in Listing 10-15 contains two procedures. The first loops through name entries, invoking a callback function for each one. The second loops through the properties for a name entry, invoking a callback function for each property. It is up to the caller to determine what the callback functions will do, but they could (for example) display a graph of names and properties in a window or identify all name entries that match a complex set of search criteria.

Listing 10-15 Listing names and properties

```
#include <NameRegistry.h>

OSStatus JoePro_ListDevices(
    void (*callback) (
        RegCStrPathName         *name,
        RegEntryID              *entry
        )
    )
{
    OSErr err = noErr;
    RegEntryIter cookie;
    Boolean done;

    /*
     * Entry iterators are created pointing to the root
     * with a RegEntryIterationOp of kRegIterDescendants.
     * So, we just need to continue.
     */
    RegEntryIterationOp op = kRegIterContinue;

    err = RegistryEntryIterateCreate(&cookie);
    if (err == noErr) do {
        RegEntryID entry;

        err = RegistryEntryIterate(&cookie, op, &entry, &done);
        if (!done) {
            RegCStrPathName         *name;
            RegPathNameSize         len;

            err = RegistryCStrEntryToPathSize(&entry, &len);
            if (err == noErr) {
                name = (RegCStrPathName*) malloc(len);

                assert(name != NULL);

                err = RegistryCStrEntryToPath(&entry, name, len);
                if (err == noErr) {
                    (*callback)(name, &entry);
                }
                free(name);
            }
            RegistryEntryIDDispose(&entry);
        }
    } while (!done);
    RegistryEntryIterateDispose(&cookie);
    return err;
}

OSStatus JoePro_ListProperties(
    const RegCStrPathName           *name,
    const RegEntryID                *entry,
    void            (*callback)(
                     RegPropertyName*,
                     RegPropertyValue,
                     RegPropertyValueSize
                     )
    )
```

```
{
    OSErr err = noErr;
    RegPropertyIter cookie;
    Boolean done;

    err = RegistryPropertyIterateCreate(entry, &cookie);
    if (err == noErr) do {
        RegPropertyNameBuf          property;

        err = RegistryPropertyIterate(&cookie, property, &done);
        if (!done) {
            RegPropertyValue            val;
            RegPropertyValueSize        siz;

            err = JoePro_GetProperty(entry, property, &val, &siz);
            if (err == noErr) {
                (*callback)(property, val, siz);
            }
        }
    } while (!done);
    RegistryPropertyIterateDispose(&cookie);
    return err;
}
```

C H A P T E R   1 1

Driver Services Library

This chapter describes the routines that are provided for every native driver by the Mac OS Driver Services Library. The routines included in the Driver Services Library implement all the programming interfaces that Mac OS provides for drivers. Additional functionality may be made available to drivers within certain families or categories through family programming interfaces (FPIs) maintained by family experts.

As described in the next section, device drivers run in their own environment without access to the Mac OS Toolbox. This chapter describes the services available in the device driver run-time environment. The services are categorized as follows:

- memory management
- interrupt management
- timing services
- atomic operations
- queue operations
- string operations
- debugging support
- service limitations

These services are also available to family drivers to support their basic needs. Mac OS provides some added family-specific services that are not discussed in this chapter. For further information about family-specific services, see Chapters 13 through 15.

Device Driver Execution Contexts

As explained in "Noninterrupt and Interrupt-Level Execution" (page 149), code in PCI-based Macintosh computers may run in any of three execution contexts:

- Hardware interrupt level is the execution context provided to a device driver's interrupt handler. Hardware interrupts occur as a direct result of a hardware interrupt requests. Page faults are not allowed at this context. Hardware interrupt level is also known as *primary interrupt level.*

- Secondary interrupt level is the execution context similar in concept to the previous Mac OS deferred task environment, which allows drivers to defer complex processing in order to minimize interrupt latency. Page faults are not allowed at this context.
- Noninterrupt level, usually called *task level,* is the context where all other code is executed. Page faults are allowed at this context.

Additional information about interrupts contexts can be found in Technote 1104, "Interrupt-Safe Routines." Technotes are available on the Developer CD and at the Apple Developer web site.

Note
Many device driver services are available in only one or two of the execution contexts just listed. It is the responsibility of the driver writer to conform to these limitations. Drivers that violate them will not work with future releases of Mac OS. For lists of service availability, see "Service Limitations" (page 434). ◆

CurrentExecutionLevel

The function `CurrentExecutionLevel` lets code determine its execution context.

```
ExecutionLevel CurrentExecutionLevel (void);
```

DESCRIPTION

`CurrentExecutionLevel` returns one of the result codes shown below.

EXECUTION CONTEXT

`CurrentExecutionLevel` may be called from task level, software interrupt level, or hardware interrupt level.

RESULT CODES

`kTaskLevel`	0	Noninterrupt level
`kSecondaryInterruptLevel`	5	Secondary interrupt level
`kHardwareInterruptLevel`	6	Hardware interrupt level

Miscellaneous Types

This section introduces some basic data types that are used throughout the Driver Services Library.

```
typedef unsigned long          ByteCount;

typedef unsigned long          ItemCount;

typedef long                   OSStatus;

typedef unsigned long          OptionBits;
```

For a description of `OSStatus`, see “Error Returns” (page 156).

The constant `kNilOptions` (= 0) is provided for clarity.

IDs are used whenever you create, manipulate, or destroy a object. All IDs are derived from the type `KernelID`:

```
typedef struct OpaqueRef *KernelID;
```

You should use the derived ID types whenever possible to make your code more readable.

Note

Derived ID types are all 32-bit opaque identifiers that specify various kernel resources. There is a separate ID type for each kind of resource—for example, separate types for `TaskID` and `AddressSpaceID`. All kernel services that create or allocate a resource return an ID; the ID is later used to specify the resource to perform operations on it or delete it. These IDs are `opaque` because the value of the ID tells you nothing—you can't tell from an ID which resource it identifies without calling the kernel, you can't tell what ID you'll get back the next time you create a resource, and you can't tell the relationship between any two resources by the relationship between their IDs. When a resource is deleted, its ID usually becomes invalid for a long time. This helps your code catch errors, because if you accidentally use an ID for a resource that has been deleted, chances are you'll get an error instead of doing something to a different resource. ◆

The value `kInvalidID` (= 0) is reserved to mean no ID.

Memory Management Services

This section describes the memory management services that the Driver Services Library provides to drivers.

Addressing

Current versions of the Mac OS provide a single address space that is used by all software. Future versions of Mac OS may provide memory protection and separate address spaces for different software entities. The services described in this chapter are designed to be compatible with multiple address spaces, and drivers using these services may eventually run in a multiple address space environment.

One concept that applies to multiple address spaces is that of *static logical mapping,* the ability to address client buffers logically regardless of the current address space. Static logical mapping is important because drivers in a multiple address space environment cannot depend on the client buffer's logical address to remain directly accessible for the duration of an I/O operation.

Another concept that applies to multiple address spaces is that of *memory protection,* the ability to prevent inadvertent access to data. Drivers must respect the protection of client buffers, even though they may access the buffers through means such as hardware direct memory access.

Note
Restrictions on the execution contexts in which memory allocation and deallocation services can be used are given in "Service Limitations" (page 434). ◆

I/O Operations and Memory

Several aspects of the operating system, the main processor, cache memory, and the memory hardware must be coordinated when an I/O operation is performed between an external device and a buffer in system memory:

- *Memory protection:* The I/O operation must not violate the access restrictions of
the buffer.
- *Residency:* The I/O operation must not generate unsafe page faults when accessing the buffer. Typically a buffer must have physical memory assigned to it for the duration of the I/O operation.
- *Addressability:* When using DMA hardware to perform an I/O operation, it is necessary to convert a logical buffer specification into a physical specification. When using programmed I/O, it is necessary to convert the buffer specification (either logical or physical) to a logical specification that is addressable regardless of the current address space.
- *Memory coherency:* Coherency ensures that the data being moved is not stale and that the effects of the data movement are apparent to the processor and any associated data caches. Guaranteed coherency potentially applies to cache operations before and after the I/O operation.

The DSL provides services that ensure this coordination. One service assigns physical memory to the buffer, generates an appropriate buffer specification, and performs all necessary cache manipulations prior to the I/O operation. Another routine cleans up following the I/O operation. These services operate according to the computer's cache topology, taking into account whether the caches are logical or physical and whether the overall hardware architecture guarantees coherency. This shields drivers from having to compensate for the system memory architecture.

Memory Management Types

This section defines some types and values that are fundamental to memory management for native drivers.

Values of type `LogicalAddress` represent a location in an address space:

```
typedef void *LogicalAddress;
```

Values of type `PhysicalAddress` represent location in physical memory. They are used primarily with DMA I/O operations:

```
typedef void *PhysicalAddress;
```

A `LogicalAddressRange` structure is a description of a single logically addressed buffer:

```
struct LogicalAddressRange
{
    LogicalAddress          address;
    ByteCount               count;
};

typedef struct LogicalAddressRange LogicalAddressRange;
typedef struct LogicalAddressRange *LogicalAddressRangePtr;
```

A `PhysicalAddressRange` structure is a description of a single physically addressed buffer:

```
struct PhysicalAddressRange
{
    PhysicalAddress         address;
    ByteCount               count;
};

typedef struct PhysicalAddressRange PhysicalAddressRange;
typedef struct PhysicalAddressRange *PhysicalAddressRangePtr;
```

An `AddressRange` structure is a description of a single buffer, in which the buffer address may be either logical or physical:

```
struct AddressRange
{
    void            *base;
    ByteCount       length;
};

typedef struct AddressRange AddressRange;
```

Address spaces are referred to by values of type `AddressSpaceID`. The value `kCurrentAddressSpaceID` refers to the current address space:

```
typedef KernelID AddressSpaceID;
enum
{
    kCurrentAddressSpaceID = 0
};
```

Memory Services Used During I/O Operations

The DSL provides two routines that help drivers coordinate I/O software with system memory:

- The `PrepareMemoryForIO` function tells Mac OS that a particular buffer will be used for I/O transfers. It checks memory protection, assigns physical memory to the buffer, provides addressing information, and prepares the processor's caches for the transfer.
- The `CheckpointIO` function tells the operating system that the previously started transfer is complete. It assures processor cache coherency and either prepares for further transfers or, if its parameters specify that no more transfers will be made, deallocates the resources associated with the buffer preparation. Once the preparation's resources have been deallocated, subsequent I/O operations with the buffer must be preceded by another call to `PrepareMemoryForIO`.

The memory coordination that these routines provide is summarized in "I/O Operations and Memory" (page 350).

▲ **WARNING**
Failure to use these I/O related services properly can result in data corruption or fatal system errors. Correct system behavior is the responsibility of the operating system and all I/O components including hardware, drivers, and other software. ▲

Preparing Memory for I/O

This section describes the `PrepareMemoryForIO` function and its associated data structures. Different ways of employing `PrepareMemoryForIO` are discussed in "Using PrepareMemoryForIO" (page 360).

PrepareMemoryForIO Data Structures

The `PrepareMemoryForIO` function has a single parameter, a pointer to an `IOPreparationTable` structure.

Some fields of the `IOPreparationTable` structure contain pointers to subsidiary structures. There are three types of subsidiary structures:

- A `LogicalMappingTablePtr` value is a pointer to an array of `LogicalAddress` values. The `LogicalAddress` table is where `PrepareMemoryForIO` returns the static logical addresses the driver can use to logically access the client buffer:

  ```
  typedef LogicalAddress *LogicalMappingTablePtr;
  ```

- A `PhysicalMappingTablePtr` value is a pointer to an array of `PhysicalAddress` values. The `PhysicalAddress` table is where `PrepareMemoryForIO` returns the physical addresses the driver can use to access the client buffer physically:

  ```
  typedef PhysicalAddress *PhysicalMappingTablePtr;
  ```

- An `AddressRangeTablePtr` value is a pointer to an array of `AddressRange` specifications. All ranges in a given `AddressRange` array are of the same kind, either all logical or all physical. The `AddressRange` table is where the driver can specify a user buffer that consists of multiple ranges (that is, a scatter-gather buffer as described in "Scatter-Gather Client Buffers" (page 363):

  ```
  typedef struct AddressRange *AddressRangeTablePtr;
  ```

The IOPreparationTable structure and its subsidiary structures are diagrammed in Figure 11-1 (page 356).

Note
In Figure 11-1, gray areas are filled in by the PrepareMemoryForIO function and white areas are filled in by the calling software. The preparationID field is used both ways. ◆

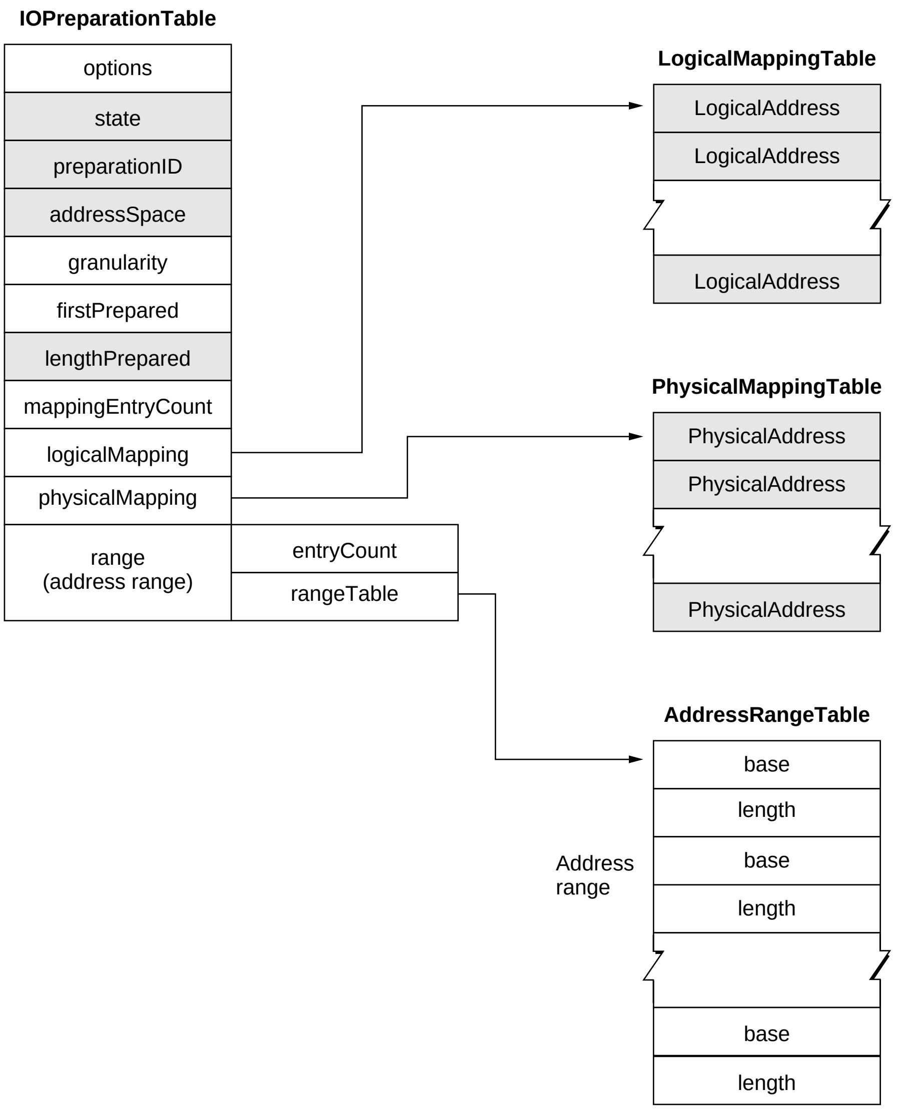
The IOPreparationTable structure is defined as follows:

```
struct IOPreparationTable
{
    IOPreparationOptions            options;
    IOPreparationState              state;
    IOPreparationID                 preparationID;
    AddressSpaceID                  addressSpace;
    ByteCount                       granularity;
    ByteCount                       firstPrepared;
    ByteCount                       lengthPrepared;
    ItemCount                       mappingEntryCount;
    LogicalMappingTablePtr          logicalMapping;
    PhysicalMappingTablePtr         physicalMapping;
    union
    {
    AddressRange                    range;
    MultipleAddressRange            multipleRanges;
    }                               rangeInfo;
};

typedef struct IOPreparationTable IOPreparationTable;

typedef OptionBits IOPreparationOptions;
enum {
    kIOMultipleRanges                   = 0x00000001,
    kIOLogicalRanges                    = 0x00000002,
    kIOMinimalLogicalMapping            = 0x00000004,
    kIOShareMappingTables               = 0x00000008,
    kIOIsInput                          = 0x00000010,
    kIOIsOutput                         = 0x00000020,
    kIOCoherentDataPath                 = 0x00000040,
    kIOClientIsUserMode                 = 0x00000080
};
```

Figure 11-1 `IOPreparationTable` structure



```
typedef OptionBits IOPreparationState;
enum {
    kIOStateDone                    = 0x00000001
};

typedef struct MultipleAddressRange MultipleAddressRange;

struct MultipleAddressRange
{
    ItemCount                       entryCount;
    AddressRangeTablePtr            rangeTable;
};
```

The `IOPreparationTable` structure specifies the buffer to be prepared and provides storage for the mapping and other information that are returned. Its fields contain the following information:

`options`	Optional characteristics of the `IOPreparationTable` structure and the transfer process. Possible values in this field are discussed in "IOPreparationTable Options" (page 358).
`state`	Filled in by `PrepareMemoryForIO` to indicate the state of the `IOPreparationTable` structure. The `kIOStateDone` flag indicates that the buffer has been prepared up to the end of the specified range. See "Partial Preparation" (page 365).
`preparationID`	Filled in by `PrepareMemoryForIO` to indicate the identifier that represents the I/O transaction. When the I/O operation is completed or abandoned, the `IOPreparationID` value is used to finish the transaction, as described in "Finishing I/O Transactions" (page 366).
`addressSpace`	The address space containing the buffer to be prepared. Current versions of the Mac OS provide only one address space, which it automatically passes to native drivers through `DoDriverIO`. In general, a driver should always pass the address space it received as a parameter to its `DoDriverIO` routine in this field. Otherwise, this field must be specified as `kCurrentAddressSpaceID`.
`granularity`	Information to reduce the memory usage of partial preparations. See "Partial Preparation" (page 365).
`firstPrepared`	The byte offset into the buffer at which to begin preparation. See "Partial Preparation" (page 365).

`lengthPrepared`	Filled in by `PrepareMemoryForIO` to indicate how much of the buffer was successfully prepared, beginning at `firstPrepared`. See "Partial Preparation" (page 365).
`mappingEntryCount`	Number of entries in the logical and physical mapping tables supplied. Normally, the driver should allocate as many entries as there are pages in the buffer. The number of pages in a memory range can be calculated from the range's base address and length. If there are not enough entries, a partial preparation is performed within the limit of the tables. See "Partial Preparation" (page 365).
`logicalMapping`	The address of an array of `LogicalAddress` values. `PrepareMemoryForIO` fills the logical mapping table with the static logical mappings for the specified buffer. This table is optional. Mapping tables are discussed in "Mapping Tables" (page 362).
`physicalMapping`	The address of an array of `PhysicalAddress` values. `PrepareMemoryForIO` fills the physical mapping table with the physical addresses corresponding to the specified buffer. This table is optional. Mapping tables are discussed in "Mapping Tables" (page 362).
`rangeInfo`	The buffer to prepare. A simple buffer is represented by a single `AddressRange` value. A scatter-gather buffer is specified by a `MultipleAddressRange` structure. If the `kIOMultipleRanges` flag is omitted from `options`, `rangeInfo` is interpreted as an `AddressRange` value named `range`. If `kIOMultipleRanges` is specified in `options`, `rangeInfo` is interpreted as a `MultipleAddressRange` structure named `multipleRanges`. Scatter-gather buffers are discussed in "Scatter-Gather Client Buffers" (page 363). Because there might be insufficient resources to prepare the entire buffer, the buffer can be prepared in pieces. This procedure is discussed in "Partial Preparation" (page 365).

IOPreparationTable Options

This `options` field of the `IOPreparationTable` structure contains flags that have the following meanings:

- `kIOMultipleRanges` specifies that the `rangeInfo` field is to be interpreted as `MultipleAddressRange`, enabling a scatter-gather memory specification.

- `kIOLogicalRanges` specifies that the base fields of the `AddressRange` structures are logical addresses. If this option is omitted, the addresses are treated as physical addresses. Current versions of the Mac OS do not support specifying physical buffers, so the driver must always specify this option.
- `kIOMinimalLogicalMapping` specifies that the `LogicalMappingTable` structure is to be filled in with just the first and last mappings of each range, arranged in pairs. Minimal logical mappings are discussed in "DMA Alignment Requirements" (page 364).
- `kIOShareMappingTables` specifies that the system can use the driver's mapping tables instead of maintaining its own copies of the tables. Sharing mapping tables is discussed in "Reducing Memory Usage" (page 363).
- `kIOIsInput` specifies that data will be moved into main memory.
- `kIOIsOutput` specifies that data will be moved out of main memory.
- `kIOCoherentDataPath` indicates that the data path that will be used to access memory during the I/O operation is fully coherent with the main processor's data caches, making data cache manipulations unnecessary. Memory coherency with the instruction cache is not implied, however, so the appropriate instruction cache manipulations are performed regardless. This option is useful when the overall hardware architecture is not coherent, but the driver knows that the transfer will occur on a particular hardware path that is coherent. (`PrepareMemoryForIO` operates according to the overall architecture and has no implicit way of knowing about individual data paths.) When in doubt, omit this option. Incorrectly omitting it merely slows operation, whereas incorrectly specifying this option can result in erroneous behavior and crashes.
- `kIOClientIsUserMode` indicates that `PrepareMemoryForIO` is being called on behalf of a nonprivileged client. If this option is specified, the memory ranges are checked for user-mode accessibility. If this option is omitted, the memory ranges are checked for privileged-level accessibility. Drivers can obtain the client's execution mode through the device's family programming interface (FPI). This option is not implemented in current versions of Mac OS. For compatibility with future Mac OS releases, drivers should omit it from the options. The FPI will perform the buffer access level checks on behalf of the driver.

Using PrepareMemoryForIO

`PrepareMemoryForIO` coordinates data transfers between devices and one or more memory ranges in the system, the main processor caches, and other memory facilities. Preparation includes ensuring that physical memory remains assigned to the memory ranges until `CheckpointIO` relinquishes it. Depending on the I/O direction and data path coherence that are specified, Mac OS manipulates the contents of the processor's caches, if any, and may make parts of physical memory noncacheable.

PrepareMemoryForIO

```
OSStatus

PrepareMemoryForIO (IOPreparationTable *theIOPreparationTable);
```

`--> theIOPreparationTable`
Pointer to an `IOPreparationTable` structure

DESCRIPTION

`PrepareMemoryForIO` coordinates data transfers between devices and one or more memory ranges with the operating system, the main processor caches, and other data buffers. Preparation includes ensuring that physical memory remains assigned to the memory ranges until `CheckpointIO` relinquishes it. Depending on the I/O direction and data path coherence that are specified, Mac OS manipulates the contents of the processor's caches, if any, and may make parts of the ranges noncacheable.

A native driver can call `PrepareMemoryForIO` from its `DoDriverIO` handler. The `DoDriverIO` entry point is discussed in "DoDriverIO Entry Point" (page 204).

The driver or other software must perform I/O preparation before permitting data movement. For operations with block-oriented devices, preparation is best done just before moving the data, typically by the driver. For operations upon buffers such as memory shared between the main processor and a coprocessor, frame buffers, or buffers internal to a driver, preparation is best performed when the buffer is allocated. This technique is discussed more fully in "Multiple Transfers" (page 363). The PCI Card Device Driver Kit contains code samples that use `PrepareMemoryForIO`.

Calls to `PrepareMemoryForIO` should be matched with calls to `CheckpointIO`, even if the I/O operation was aborted. In addition to applying finishing operations to the memory range, `CheckpointIO` deallocates resources used in preparing the range.

The `IOPreparationTable` must remain allocated until the last `CheckPointIO` is called.

EXECUTION CONTEXT

`PrepareMemoryForIO` may be called only at task level from a driver's `DoDriverIO` routine or from a subroutine called by `DoDriverIO`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Logical and Physical Memory Preparation

The two most common `PrepareMemoryForIO` operations are preparing logical or physical I/O when the client has specified a single, logically-addressed buffer. The following lists show how the driver would set up the `IOPreparationTable` for these cases. The only difference between the two cases is which mapping table is supplied. `PrepareMemoryForIO` infers whether the transfer will be physical (DMA) or logical (programmed I/O) based on whether the mapping table is physical or logical.

To perform logical I/O with single logical buffer, set `IOPreparationTable` as follows:

`options`	`kIOLogicalRanges` and either `kIOIsInput` or `kIOIsOutput`
`addressSpace`	address parameter used in the `DoDriverIO` call (page 357)
`granularity`	0
`firstPrepared`	0
`mappingEntryCount`	Number of pages in buffer
`logicalMapping`	Address of table containing `mappingEntryCount` entries

`physicalMapping`	`nil`
`range.base`	Buffer address
`range.length`	Buffer length

For physical I/O with single logical buffer, set `IOPreparationTable` as follows:

`options`	`kIOLogicalRanges` and either `kIOIsInput` or `kIOIsOutput`
`addressSpace`	address parameter used in the `DoDriverIO` call
`granularity`	0
`firstPrepared`	0
`mappingEntryCount`	Number of pages in buffer
`logicalMapping`	`nil`
`physicalMapping`	Address of table containing `mappingEntryCount` entries
`range.base`	Buffer address
`range.length`	Buffer length

Mapping Tables

The logical and physical mapping tables are where `PrepareMemoryForIO` returns the addresses the driver can use to access the client's buffer. The first entry of a range's mappings will be the exact mapping of the first prepared address in that range, regardless of page alignment, while the remaining entries will be page aligned. If multiple address ranges were specified, the mapping table is a concatenation, in order, of the mappings for each range.

There are no explicit length fields in the mapping tables. Instead, entry lengths are implied by the entry's position in the range's mappings, the overall range length, and the page size. The length of the first entry generally runs to the next page alignment, the length of the intermediate entries (if any) is the page size, and the length of the last element in the range is what remains by subtracting the previous lengths from the overall range length. If the prepared range fits within a single page, there is only one prepared entry and its length is equal to the range length.

Scatter-Gather Client Buffers

Drivers may be asked to transfer data from buffers that are not contiguous. In this case, the client buffer may be specified as a `MultipleAddressRange` scatter-gather list.

A `MultipleAddressRange` structure specifies an array of `AddressRange` entries. Its fields have the following meanings:

`entryCount`	The number of entries in the `rangeTable` structure.
`rangeTable`	The address of an array of `AddressRange` elements (an `AddressRangeTable` structure). See the description of `AddressRange` in "PrepareMemoryForIO Data Structures" (page 353). The specified ranges may overlap.

The `options` and `addressSpace` specifications apply equally to each range.

The `granularity`, `firstPrepared`, and `lengthPrepared` fields apply to the overall buffer. These fields are discussed in "Partial Preparation" (page 365).

The resulting mapping tables concatenate, in order, the mappings for each range.

Multiple Transfers

This DSL memory management services allow efficient coordination for both single and multiple I/O transactions to a given buffer. A single transaction—such as reading page-faulted data into a client's memory—uses a `PrepareMemoryForIO` call before the transfer and a single `CheckpointIO` call when the transfer is complete. A multiple transaction scenario—such as a network driver that transfers from its own buffers and divides blocks in and out of the client buffer—uses a single `PrepareMemoryForIO` call during driver initialization and a `CheckpointIO` call before and after each transfer. The intermediate calls to `CheckpointIO` would include the `kIOMoreTransfers` option, so the memory preparation remains in effect.

Reducing Memory Usage

`PrepareMemoryForIO` normally keeps its own copy of the mapping tables in addition to the tables the driver has allocated. Hence, memory usage can be reduced if the driver shares its mapping tables with the operating system. The `kIOShareMappingTables` option specifies that `PrepareMemoryForIO` can use the driver's mapping tables rather than maintain its own copies. The shared mapping tables must be located in logical memory that cannot page fault until the final `CheckpointIO` call finishes (that is, the memory is locked). In addition, the mapping tables must remain allocated and the entries unaltered until after the final `CheckpointIO` call. It is not necessary for the driver to provide both tables.

A full-sized mapping table contains as many entries as there are pages in the client buffer. However, the driver can use a smaller table if it calls `PrepareMemoryForIO` more than once for a given client buffer. This technique is discussed in "Partial Preparation" (page 365).

The `granularity` specification can reduce memory usage in the event of a partial preparation. Granularity is discussed in "Partial Preparation" (page 365).

Certain DMA transactions require both mapping tables. However, the size of the logical mapping table can be easily reduced. The `kIOMinimalLogicalMapping` option is discussed in "DMA Alignment Requirements" (page 364).

Reducing Execution Overhead

If memory must be prepared long in advance of the transfer, the driver can reduce the execution overhead by postponing cache manipulations. This is because cache manipulations are wasted if the buffer will be accessed normally before the transfer. By omitting both `kIOIsInput` and `kIOIsOutput` from the `options` field, the driver prevents `PrepareMemoryForIO` from manipulating the caches at that time. Later, the driver calls `CheckpointIO` just prior to the transfer to prepare the caches. This is part of the technique discussed in the "Multiple Transfers" (page 363).

DMA Alignment Requirements

A variation on the physical transfer of data occurs when the client's buffer does not meet the alignment requirements of the DMA hardware. In this case, the driver needs to supply a logical mapping table in addition to the physical mapping table, so that programmed I/O can be performed in the unaligned beginning and/or end of the buffer. Otherwise, the driver would have to prepare the beginning and end separately from the middle portion.

Because only the beginning and the end of the buffer will be transferred with programmed I/O, only the first and last logical mapping table entries are actually needed—the middle entries are page aligned, which is usually sufficient for any DMA engine. To reduce memory usage, the driver may limit the size of the logical mapping table to just two entries per range and may specify the kIOMinimalLogicalMapping option. PrepareMemoryForIO will fill in the first logical mapping table entry of each range as usual and will fill the second entry with the static logical mapping of the last page in the range. Two entries per range are used, regardless of the range sizes. However, the value of the second entry of the pair is undefined if the range is contained within a single page.

Partial Preparation

If insufficient resources are available to prepare the whole range of memory that is specified, PrepareMemoryForIO will prepare as much as possible, indicate to the driver how much memory was prepared, clear the kIOStateDone bit in tableState, and return noErr. This is called a *partial preparation.*

Examples of resources that may limit the preparation are insufficient physical page frames to make the buffer resident, mapping table size too small, and not enough operating-system pool space. Because not all of these resources are under the control of the driver, every driver that calls PrepareMemoryForIO must be written to handle a partial preparation. One possibility is to make the terminating CheckpointIO call to deallocate the preparation's resources and return an error to the client. Another possibility is to perform the transfer as a series of partial transfers.

The firstPrepared, lengthPrepared, and granularity fields of the IOPreparationTable structure (shown in Figure 11-1 (page 356)) control partial preparations. When calling PrepareMemoryForIO the first time, specify 0 for firstPrepared. If the resulting tableState value does not indicate kIOStateDone, a partial preparation was performed, and lengthPrepared indicates how much memory was successfully prepared. After the data transfer and final call to CheckpointIO, another PrepareMemoryForIO call can be made to prepare as much as possible of the ranges that remain. This time, firstPrepared should be the sum of the current firstPrepared and lengthPrepared. This sequence prepare, transfer, and final checkpoint can be repeated until IOPreparationState indicates kIOStateDone.

The granularity field gives a hint to PrepareMemoryForIO for partial preparation. It is useful for transfers with devices that operate on fixed-length buffers. The length prepared will be 0 (with an error status returned) or a multiple of granularity rounded up to the next greatest page alignment. This prevents preparing more memory than the driver is willing to use. A value of 0 for granularity specifies no granularity. No check is made for whether the specified range lengths are multiples of granularity.

Finishing I/O Transactions

This section describes the `CheckpointIO` function and its options.

CheckpointIO

```
OSStatus CheckpointIO(
                    IOPreparationID theIOPreparation,
                    IOCheckpointOptions theOptions);
```

`--> theIOPreparation`
Value from the `IOPreparationID` field in the `IOPreparationTable` structure.

`--> theOptions`
Options value, as defined in `OptionBits`.

```
typedef OptionBits IOPreparationOptions;
enum{
    kNextIOIsInput          = 0x00000001,
    kNextIOIsOutput         = 0x00000002,
    kMoreIOTransfers        = 0x00000004
};
```

DESCRIPTION

`CheckpointIO` performs the necessary follow-up operations for a device I/O transfer and optionally prepares for a new transfer or reclaims the system resources associated with memory preparation. To reclaim resources, `CheckpointIO` should be called even if the I/O operation was abandoned.

Mac OS supports multiple concurrent preparations of memory ranges or portions of memory ranges. In this case, cache actions are appropriate and individual pages are not unlocked until all transactions have been finalized.

The `theIOPreparation` parameter is the `IOPreparationID` value for the I/O operation, as returned by a previous call to `PrepareMemoryForIO`. This ID is not valid following `CheckpointIO` if the `kMoreTransfers` option is omitted.

The `Options` parameter specifies optional operations. Values for this field are the following:

`kNextIOIsInput`	Data will be moved into main memory.
`kNextIOIsOutput`	Data will be moved out of main memory.
`kMoreIOTransfers`	Further I/O transfers will occur to or from the buffer. If `kMoreIOTransfers` is omitted, the buffer is allowed to page and `IOPreparationID` is invalidated.

The `IOPreparationTable` must remain allocated until the last `CheckPointIO` is called.

EXECUTION CONTEXT

`CheckpointIO` may be called from task level or secondary interrupt level but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Cache Operations

Unlike some previous Macintosh drivers, native drivers do not need to flush the PowerPC processor cache in the case of normal I/O operations. The Power Macintosh hardware supports processor cache snooping, which guarantees that the RAM and cache memory domain is coherent. The `PrepareMemoryForIO` routine takes care of maintaining cache coherency inherent in PCI-based Macintosh computers.

Nevertheless, driver writers may want to perform cache manipulation to improve driver performance. The Driver Services Library provides several routines and data types, described in this section, that allow drivers to get information about cache, alter the default cache modes, and flush the processor cache.The `SetProcessorCacheMode` function, described on (page 372), forces the cache mode for selected pages of memory. The `FlushProcessorCache` function, described on (page 374), forces data from cache out to main memory. These functions lets special-purpose drivers optimize their I/O performance.

▲ **WARNING**
Take care when using the `SetProcessorCacheMode` and `FlushProcessorCache` functions, because they may conflict with the cache mode operations of Mac OS. Most drivers need use only `PrepareMemoryForIO` and `CheckPointIO`. ▲

Getting Cache Information

The functions described in this section let you determine the structure of the processor cache. `GetLogicalPageSize` and `GetDataCacheLineSize` define the structure of the cache, and `GetPageInformation` returns information about each logical page in an address range.

GetLogicalPageSize

```
ByteCount GetLogicalPageSize (void);
```

DESCRIPTION

The `GetLogicalPageSize` function returns the logical page size of the cache, in bytes. If you need this value often, call it once at startup and store the value in a global. The value of the logical page size of the cache cannot change until the system reboots.

EXECUTION CONTEXT

`GetLogicalPageSize` may be called from task level, software interrupt level, or hardware interrupt level.

GetDataCacheLineSize

```
ByteCount GetDataCacheLineSize (void);
```

DESCRIPTION

The `GetDataCacheLineSize` function returns the line size of the cache, in bytes.

EXECUTION CONTEXT

`GetDataCacheLineSize` may be called from task level, software interrupt level, or hardware interrupt level.

GetPageInformation

```
OSStatus GetPageInformation(
                     AddressSpaceID theAddressSpace,
                     LogicalAddress theBase,
                     ByteCount theLength,
                     PBVersion theVersion,
                     PageInformation *thePageInfo);
```

`--> theAddressSpace`
ID of address space to be examined.

`--> theBase` Starting address in address space.

`--> theLength` Length of address range, in bytes.

`--> theVersion`
Version of the page information structure.

`<-- thePageInfo`
Pointer to the page information structure.

```
struct PageInformation
{
    AreaID                     area;
    ItemCount                  count;
    PageStateInformation       information [1];
};

typedef unsigned long PageStateInformation;
enum {
    kPageIsProtected                  = 0x00000001,
    kPageIsProtectedPrivileged          = 0x00000002,
    kPageIsModified                     = 0x00000004,
    kPageIsReferenced                   = 0x00000008,
    kPageIsLocked                       = 0x00000010,
    kPageIsResident                     = 0x00000020,
    kPageIsShared                       = 0x00000040,
    kPageIsWriteThroughCached           = 0x00000080,
    kPageIsCopyBackCached               = 0x00000100
};

typedef struct PageInformation PageInformation,
                     *PageInformationPtr;
```

DESCRIPTION

The `GetPageInformation` function returns information about each logical page in a specified range. Parameter `theAddressSpace` specifies the address space containing the range of interest. Parameter `theBase` is the first logical address of interest. Parameter `theLength` specifies the number of bytes of logical address space, starting at `theBase`, about which information is to be returned.

Parameter `theVersion` specifies the version number of the `PageInformation` type to be returned, thereby providing backward compatibility.

Parameter `thePageInfo` is filled in with information about each logical page. This buffer must be large enough to contain information about the entire range. The page information fields are the following:

- The Mac OS currently sets this field to `kNoAreaID`.
- `count` indicates the number of entries in which information was returned.
- `information` contains one `PageStateInformation` entry for each logical page.

The bits of `PageStateInformation` are the following:

- `pageIsProtected`: the page is write-protected against unprivileged software.
- `pageIsProtectedPrivileged`: the page is write-protected against privileged software.
- `pageIsModified`: the page has been modified since the last time it was mapped in or its data was released.

- `pageIsReferenced`: the page has been accessed (by either a load or a store operation) since the last time the memory system's paging operation checked the page.
- `pageIsLocked`: the page is ineligible for replacement (it is nonpageable) because there is at least one outstanding `PrepareMemoryForIO` or `SetPagingMode` (of `kPagingModeResident`) request outstanding that uses it.
- `pageIsShared`: the page's underlying physical page is mapped into additional logical pages.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

EXECUTION CONTEXT

`GetPageInformation` may be called only from task level, not from secondary or hardware interrupt level.

Setting Cache Modes

Mac OS assigns default cache modes to various kinds of memory. Main memory defaults to copyback cache mode; PCI memory space defaults to cache-inhibited mode

With these settings, drivers do not need to perform specific cache flushing. However, drivers may wish to alter a memory section's default cache mode to create the highest performance data transfer rate for their application. For example, the PowerPC processor performs burst bus transactions to memory in copyback or writethrough cache modes.

Drivers may also want to set areas of PCI memory space to a cacheable setting, thereby causing the PowerPC to burst to that space; however, extreme care must be taken to perform appropriate cache flushing when operating on cacheable PCI memory space. Drivers that control PCI master devices may wish to experiment with different cache modes for their DMA buffer spaces to determine the optimal setting.

SetProcessorCacheMode

```
OSStatus SetProcessorCacheMode(
                    AddressSpaceID theAddressSpace,
                    void *theBase,
                    ByteCount theLength,
                    ProcessorCacheMode theMode);
```

`-->` `theAddressSpace`	Address space ID of address space.
`-->` `theBase`	Pointer to the starting address in address space.
`-->` `theLength`	Length of address range, in bytes.
`-->` `theMode`	Cache mode to be set, as defined in `ProcessorCacheMode`.

```
typedef unsigned long ProcessorCacheMode;
enum {
    kProcessorCacheModeDefault                = 0,
    kProcessorCacheModeInhibited              = 1,
    kProcessorCacheModeWriteThrough           = 2,
    kProcessorCacheModeCopyBack               = 3
};
```

DESCRIPTION

The `SetProcessorCacheMode` function sets the cache mode of a specified range of address space. The `theAddressSpace` parameter specifies the address space containing the logical ranges to be set. Current versions of the Mac OS provide only one address space, which it automatically passes to native drivers through `DoDriverIO`. In general, a driver should always pass the address space it received as a parameter to its `DoDriverIO` routine in this field. Otherwise, the address space must be specified as `kCurrentAddressSpaceID`.

In early versions of the PCI-based Mac OS, `SetProcessorCacheMode` can be used on only one card in any given 256 MB segment of the effective address space above 0x7FFF FFFF. For example, if two PCI cards were configured at addresses 0x8001 2000 and 0x8034 5000, `SetProcessorCacheMode` could set the cache mode of only one card's address space. However, it could also set the mode of a card at 0xA001 0000, because that card's space lies in a different 256 MB segment of the system's effective address space. This restriction will be relaxed in future versions of Mac OS.

EXECUTION CONTEXT

`SetProcessorCacheMode` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Synchronizing I/O

To synchronize I/O accesses use the `SynchronizeIO` routine. You can call it either before or after accesses—the object is simply to separate the accesses by `eieio` actions.

SynchronizeIO

```
void SynchronizeIO (void)
```

DESCRIPTION

The `SynchronizeIO` routine performs the necessary operations to ensure orderly code execution between accesses to noncached devices.

▲ **WARNING**
Failure to use `SynchronizeIO` between I/O accesses can misorder load and store operations, with unpredictable results for program execution. ▲

EXECUTION CONTEXT

`SynchronizeIO` may be called from task level, secondary interrupt level, or hardware interrupt level.

Flushing the Processor Cache

As explained in “Cache Operations” (page 367), drivers normally do not need to flush the processor cache. The function described in this section is used only in rare cases to improve performance.

FlushProcessorCache

```
OSStatus FlushProcessorCache(
                    AddressSpaceID spaceID,
                    LogicalAddress base,
                    By teCount length);
```

`-->` `spaceID`	Target address space identifier.
`-->` `base`	Starting address in address space.
`-->` `length`	Length of address range, in bytes.

DESCRIPTION

The `FlushProcessorCache` function forces data from cache out to main memory. The `spaceID` parameter specifies the address space containing the logical ranges prepared. Current versions of the Mac OS provide only one address space, which it automatically passes to native drivers through `DoDriverIO`. In general, a driver should always pass the address space it received as a parameter to its `DoDriverIO` routine in this field. Otherwise, the address space must be specified as `kCurrentAddressSpaceID`.

EXECUTION CONTEXT

`FlushProcessorCache` may be called from task level, secondary interrupt level, or hardware interrupt level.

RESULT CODES

noErr	0	No error
paramErr	–50	Bad parameter

Memory Allocation and Deallocation

The Driver Services Library provides services to allocate and free system memory for device drivers. The `PoolAllocateResident` and `PoolDeallocate` functions allocate and deallocate resident memory. `MemAllocatePhysicallyContiguous` and `MemDeallocatePhysicallyContiguous` allocate and deallocate memory that is resident and physically unbroken. You should always use these services to obtain dynamic memory.

PCI drivers that allocate memory may need to increase the size of the system heap. They can do this by adding a `'sysz'` resource to the driver resource file, thereby extending the system heap at startup. Typical code is shown in Listing 11-1.

Listing 11-1 Adding a `'sysz'` resource to the system heap

```
type 'sysz' {
    longint;
};
resource 'sysz' (0, "256 Kb") {
    256 * 1024                              /* 1/4 MB of system heap */
};
```

Memory allocations can be performed only at noninterrupt execution level. Memory deallocations can be performed at task execution level. Execution levels are discussed in "Driver Execution Contexts" (page 194).

PoolAllocateResident

```
void PoolAllocateResident(
                     ByteCount byteSize,
                     Boolean clear);
```

`--> byteSize`	The number of bytes of memory to allocate.
`--> clear`	Whether or not the allocated memory is to be zeroed.

DESCRIPTION

The `PoolAllocateResident` function allocates resident memory `byteSize` in length. The memory address is returned as the result of the call. A `nil` result indicates that the pool is exhausted.

EXECUTION CONTEXT

`PoolAllocateResident` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Queue element not found
`memFullErr`	–108	Not enough room in heap

`MemAllocatePhysicallyContiguous`

```
LogicalAddress MemAllocatePhysicallyContiguous (
                    ByteCount byteSize,
                    Boolean clear);
```

`--> byteSize`	The number of bytes of memory to allocate.
`--> clear`	Whether or not the allocated memory is to be zeroed.

DESCRIPTION

`MemAllocatePhysicallyContiguous` allocates a buffer that is resident and is guaranteed to be physically uninterrupted. It returns the buffer's logical address.

Driver code can pass the address returned by `MemAllocatePhysicallyContiguous` to `PrepareMemoryForIO` (page 360) to obtain the buffer's physical location.

EXECUTION CONTEXT

`MemAllocatePhysicallyContiguous` may be called only from task level, not from secondary or hardware interrupt level. Requesting a larger size especially anytime after your driver initially starts up is unlikely to succeed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`memFullErr`	–108	Not enough room in heap

PoolDeallocate

```
OSStatus PoolDeallocate (LogicalAddress address);
```

`--> address`	Address of pool memory chunk to deallocate.

DESCRIPTION

The `PoolDeallocate` routine returns the chunk of memory at `address` to the pool from which it was allocated. It can be used to deallocate memory that was allocated with `PoolAllocateResident`.

EXECUTION CONTEXT

`PoolDeallocate` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Queue element not found
`memFullErr`	–108	Not enough room in heap

CODE SAMPLE

The code shown in Listing 11-2 uses `PoolDeallocate` to dispose of a property that was obtained by calling the `RegistryPropertyGet` function.

Listing 11-2 Disposing of a property

```
void DisposeThisProperty(
       RegPropertyValue        *regPropertyValuePtr
   )
{
       if (*regPropertyValuePtr != NULL) {
           PoolDeallocate(*regPropertyValuePtr);
           *regPropertyValuePtr = NULL;
       }
}
```

MemDeallocatePhysicallyContiguous

```
OSStatus MemDeallocatePhysicallyContiguous(
                     LogicalAddress address);
```

`--> address`	Address of the memory block to free.

DESCRIPTION

The `MemDeallocatePhysicallyContiguous` function deallocates memory allocated by `MemAllocatePhysicallyContiguous`.

EXECUTION CONTEXT

`MemDeallocatePhysicallyContiguous` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`notLockedErr`	–623	Specified memory range is not locked

Memory Copying Routines

The DSL provides a general routine, `BlockCopy`, for copying the contents of memory from one location to another. It also provides several `BlockMove` routines that drivers may use to more precisely control the copying process and its effects on memory coherency.

BlockCopy

`BlockCopy` copies the contents of memory from one location to another.

```
void BlockCopy      (const void *srcPtr,
                     void *destPtr,
                     Size byteCount);
```

`srcPtr`	Address of source to copy.
`destPtr`	Address of destination to copy into.
`byteCount`	Number of bytes to copy.

DESCRIPTION

The `BlockCopy` routine copies the chunk of memory at `srcPtr` to `destPtr`. Parameter `byteCount` specifies how many bytes are copied.

`BlockCopy` calls `BlockMove`, using the most appropriate version for the current execution environment and copying task. However, drivers may bypass `BlockCopy` and call `BlockMove` directly.

EXECUTION CONTEXT

`BlockCopy` may be called from task level, software interrupt level, or hardware interrupt level.

BlockMove

The DSL includes new extensions to the `BlockMove` routine that deliver improved performance for software running in native mode. The original `BlockMove` routine is described in *Inside Macintosh: Memory.*

Table 11-1 lists the different versions of the `BlockMove` function that are in the DSL. It indicates for each routine what memory contents it is designed for and whether it can be used with buffers or other destinations that are not level-one cached.

Table 11-1 `BlockMove` versions

Version	Use with what memory contents	Can be used with buffers
`BlockMove`	Contains some 68K code, L1 cached	No
`BlockMoveData`	No 68K code, L1 cached (fastest)	No
`BlockMoveDataUncached`	No 68K code, uncached	Yes
`BlockMoveUncached`	Some 68K code, uncached (slowest)	Yes
`BlockZero`	Set memory to zero, L1 cached	No
`BlockZeroUncached`	Set memory to zero, uncached	Yes

DESCRIPTION

The `BlockMove` extensions provide a way to handle cache-inhibited address spaces and are able to flush the dynamic recompilation emulator's cache, and include high-speed routines for setting memory to 0.

The `BlockMove` extensions use 8-byte floating-point registers for large blocks and assume a data cache block size of 32 bytes. They may not work if the 8-byte floating point hardware is disabled or absent or if cache blocks are larger than 32 bytes. They do not use `lswx` and `stswx` instructions, which are slow on Macintosh models other than those using the PowerPC 601.

Except for `BlockZero` and `BlockZeroUncached`, the `BlockMove` extensions use the same parameters as `BlockMove`. Calls to `BlockZero` and `BlockZeroUncached` have only two parameters, a pointer and a length, which are the same as the second and third parameters of `BlockMove`.

IMPORTANT

The `BlockMove` versions for cacheable data use the PowerPC `dcbz` instruction to avoid unnecessary prefetching of destination cache blocks. For uncacheable data, you should avoid using those routines because the `dcbz` instruction faults on uncacheable or write through locations, making execution extremely slow. ▲

EXECUTION CONTEXT

The `BlockMove` routines may be called from task level, software interrupt level, or hardware interrupt level.

Interrupt Management

This section discusses interrupt management for native drivers in PCI-based Power Macintosh computers. A general description of the new interrupt model is given first, followed by a detailed description of its programming interface. Interrupt timing services are described in "Interrupt Timers" (page 422).

Definitions

A **hardware interrupt** is a physical device's method for requesting attention from a computer. The physical device capable of interrupting the computer is known as an **interrupt source.** The device's request for attention is usually asynchronous with respect to the computer's execution of code.

An **interrupt handler** is a piece of code invoked to satisfy a hardware interrupt. Interrupt handlers are installed and removed by drivers and act as subroutines of the driver. A typical interrupt handler consists of two parts: a hardware interrupt handler and a **secondary interrupt handler.** The hardware interrupt handler is the code that services the immediate needs of the device that caused the interrupt, performing actions that must be synchronized with it. The secondary interrupt handler is the code that perform the remainder of the work associated with the interrupt. Secondary interrupt handlers are executed at a lower priority than hardware interrupt handlers.

Interrupt handler **registration** is the process of associating an interrupt source with an interrupt handler. **Interrupt dispatching** is the sequence of steps necessary to invoke an interrupt handler in response to an interrupt.

Execution contexts for interrupt handling are discussed in "Noninterrupt and Interrupt-Level Execution" (page 149).

Interrupt Model

Interrupt dispatching and control hardware may be designed in a variety of styles and capabilities. In some hardware systems, software must do most of the work of determining which devices that generate interrupts need to be serviced and in what order the system must service them. Other hardware systems may contain specific vectorization and priority schemes that force the software to respond in predetermined ways.

Designing a driver so that it can respond to the details of every interrupt mechanism in every hardware system limits the driver's portability and increases its complexity. As a result, the native driver interrupt model for PCI-based Macintosh computers was introduced and replaces the traditional interrupt-handling mechanisms used in previous Macintosh computers without a PCI bus. This new model provides a more standardized execution environment for interrupt processing by using two key strategies:

- The native driver model formalizes the concept of hardware and secondary interrupt levels for processing interrupts. Hardware interrupt level execution happens as a direct result of a hardware interrupt request. Secondary interrupt level provides a way to defer noncritical interrupt processing until after all hardware interrupts have been serviced, thereby reducing hardware interrupt latency.
- The control and propagation of hardware interrupts are abstracted from the driver software. An interrupt source for a device is represented by a node in a hierarchical tree, called an **interrupt source tree (IST).** Generally the leaf nodes of the tree represent interrupt sources for devices and the parent nodes represent dispatching or demultiplexing points. This removes the need for drivers to respond in detail to hardware interrupt mechanisms; they need only contain interrupt-handling code specific to the devices they control. Driver writers no longer need to know how interrupts are multiplexed by a particular hardware platform (such as through versatile interface adapters [VIAs), or handle CPU-specific low memory interrupt vectors.

IMPORTANT

A consequence of abstracting the interrupt-handling process from its hardware implementation is that interrupt service routines (ISRs) may be called when their devices did not cause the interrupt. To minimize processing overhead, each interrupt service routine must quickly determine if it is needed and return immediately if it is not. ▲

A rule that native drivers can follow to minimize interrupt processing overhead is as follows: When a driver determines that its associated device did not generate an interrupt, return `kIsrIsNotComplete`. If a driver does not know if its device generated an interrupt, return `kIsrIsComplete`. The interrupt dispatch handler will determine if the device is still issuing and interrupt and continue searching until is finds a match to service the interrupt. A discussion about how `kIsrIsComplete` and `kIsrIsNotComplete` are used can be found in "Interrupt Dispatching" (page 386).

A more detailed description of interrupt concepts follows.

Hardware and Secondary Interrupt Levels

Hardware interrupt level execution happens as a direct result of a hardware interrupt request. To insure maximum system performance, hardware interrupt handlers perform only those actions that must be synchronized with the external device that caused the interrupt and then queue a secondary interrupt handler to perform the remainder of the work associated with the interruption. Hardware interrupt handlers must operate within the restrictions of the interrupt execution model by not causing page faults and by using a limited set of operating-system services. Those services available to hardware interrupt handlers are listed in Table 11-2 (page 435).

Secondary interrupt level is similar to the deferred task concept in previous versions of Mac OS; conceptually, it exists between the hardware interrupt level and the application level. A secondary interrupt queue is filled with requests to execute subroutines that are posted for execution by hardware interrupt handlers. These handlers need to perform certain actions, but choose to defer the execution of the actions in the interest of minimizing hardware interrupt level execution. The execution of secondary interrupt handlers is serialized. For synchronization purposes, noninterrupt level execution may also post secondary interrupt handlers for execution; they are processed synchronously from the prospective of noninterrupt level but are serialized with all other secondary interrupt handlers.

Like hardware interrupt handlers, secondary interrupt handlers must also operate within the restrictions of the interrupt execution model by not causing page faults and by using a limited set of operating-system services. Those services available to secondary interrupt handlers are listed in Table 11-2 (page 435).

Note
The execution of secondary interrupt handlers may be interrupted by hardware interrupts. ◆

When writing device drivers that handle hardware interrupts, it is important to balance the amount of processing done within the hardware and secondary interrupt handlers with that done by the driver's tasks at noninterrupt level. The driver writer should make every effort to shift processing time from hardware interrupt level to secondary interrupt level and from secondary interrupt level to the driver's job of supporting its device. Doing this allows the system to be tuned so that the driver does not seize an undue amount of processing time from applications and other drivers.

Interrupt Source Tree Composition

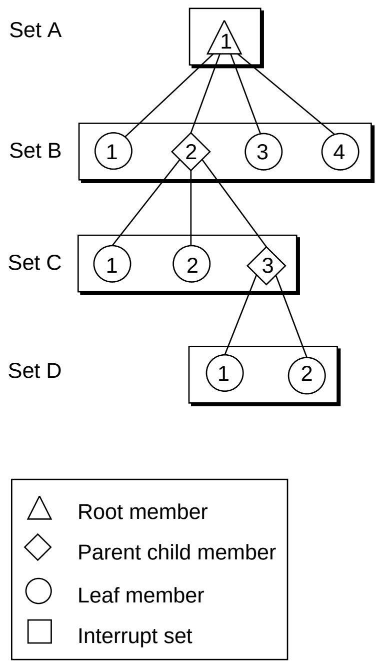
An interrupt source tree is composed of hierarchically arranged nodes. Each node represents a distinct hardware interrupt source. Nodes are called *interrupt members* and are arranged in *interrupt sets.*

An **interrupt set** is identified by an `InterruptSetID` value and is characterized as the logical grouping of all of the direct child nodes of a parent node. An `InterruptSetID` value has no meaning other than being unique among all `InterruptSetID` values. An interrupt member is identified by an `InterruptMemberNumber` value, which lies in the range from 1 to the number of members in the interrupt set to which the interrupt member belongs. Together, an `InterruptSetID` and `InterruptMemberNumber` group form an `InterruptSetMember` identifier that uniquely identifies a node in the IST.

Each interrupt set in the hierarchy represents a finer categorization of an interrupt source. The top of the tree consists of a single interrupt member that has no parent members and is referred to as the *root member.* The rest of the interrupt members in the tree branch down from the root member with each interrupt member acting as a *child member* to the interrupt members above it, and as a *parent member* to the interrupt members below it. When an interrupt member has no child members, it is referred to as a *leaf member.*

An interrupt source tree can have any number of branches, and any branch can have any number of levels. Figure 11-2 illustrates a simplified example of an interrupt source tree.

Figure 11-2 Interrupt source tree example



Interrupt Registration

An interrupt member (a node in the IST) can have four kinds of information attached
to it:

- a pointer to an interrupt service routine (ISR)
- a pointer to an interrupt enabler routine (IER)

- a pointer to an interrupt disabler routine (IDR)
- a reference constant (`refCon`)

Installation of this information is done by drivers and I/O experts during initialization. The process of attachment is called **registration.** Once registered to an interrupt member, the information persists until the next system startup.

There are two types of ISRs. The first type, called a **transversal ISR**, routes interrupt processing from a member to one of its child members. Transversal ISRs are always attached to root or parent/child members. The second type of ISR directly handles a device's request for service. This type, called a *handler ISR,* is always attached to a leaf member. Transversal ISRs never directly handle a device's request for service, and handler ISRs never directly route the processing of an interrupt.

When a handler ISR is invoked, it is supplied with three parameters. The first parameter indicates the source of the interrupt and consists of an `InterruptSetID` and `InterruptMemberNumber`, forming the `InterruptSetMember` parameter. This allows a single ISR that has been registered with multiple interrupt sources to determine which source caused the current interrupt. The second parameter is the reference constant value that was registered along with the ISR. The reference constant is not used by the system; its use is completely up to the driver writer. The third parameter is a numeric value that tells an ISR whether it has been invoked more than once in a single interrupt tree traversal process. See "InterruptHandler" (page 397) for more information.

An IER turns on an interrupt source's ability to generate a hardware interruption. Enabling a root member or parent/child member also allows any pending interrupt requests from any hierarchically lower child to propagate.

An IDR turns off an interrupt source's ability to generate a hardware interruption. It returns the previous state of the interrupt source (enabled or disabled), which can be used to decide if subsequent enable operations are required. Disabling a root member or parent/child member also prevents any pending interrupt requests from any hierarchically lower child from propagating.

Interrupt Dispatching

ISRs do all of the actual processing to service a hardware interrupt. When a device generates a hardware interrupt request, the interrupt dispatching process designates the root member of the IST the *current parent member* and invokes its ISR routine. The ISR decides which of the root member's child members should be designated as the current parent member for continued categorization of the interrupt and returns the `InterruptMemberNumber` value of that child member. As each subsequent child member is designated as the current parent member, its ISR is invoked to decide which of its child members should next be designated in the same way. Ultimately a leaf member is reached, which represents the specific interrupt source. When the leaf member's ISR is invoked, it services the specific requesting interrupt source. It then signals that processing for the interrupt is completed by returning the `kIsrIsComplete` constant. If there is no ISR attached to the leaf member, the interrupt request is dismissed as a spurious interrupt and system error is returned.

Consider an example using the simplified IST diagrammed in Figure 11-2 (page 385). Assume that the interrupt source represented by the IST member set D, `InterruptMemberNumber` value 1, requests an interruption. Interrupt dispatching begins by invoking the ISR of member set A, `InterruptMemberNumber` value 1, which returns an `InterruptMemberNumber` value of 2. This invokes the ISR of member set B, `InterruptMemberNumber` value 2, which returns an `InterruptMemberNumber` value of 3. The ISR of member set C, `InterruptMemberNumber` 3 is then invoked, and it returns an `InterruptMemberNumber` of 1. Finally, the ISR of IST member set D, `InterruptMemberNumber` 1, is invoked, which tries to service the requesting device. The ISR returns `kIsrIsComplete` if the device was successfully serviced and `kIsrIsNotComplete` if it was not successfully serviced.

Mac OS expects an ISR to return a value of `kIsrIsComplete` if it believes that the hardware associated with the ISR caused, or may have caused the interrupt. Only if an ISR is certain that its hardware did not cause the interrupt, should it return a value of `kIsrIsNotComplete`. This is true for all ISRs, unless a driver has created its own sub-tree that would allow the children of that tree to return whatever is desired by the parent node in that tree. The parent of that sub-tree is still required to follow the aforementioned rules regarding what the Mac OS expects from an ISR.

At this point the dispatching process is not complete; the tree must now be traversed back to the root. This must be done because each interrupt member set can have dispatching options attached to the set that modifies dispatching behavior. Once a leaf member's ISR has been invoked, the traversal path must be retraced toward the root to see if any parent members on the path belong to an interrupt set with dispatching options. These options can take two forms:

- reinvoke a child's parent ISR function when the child member returns `kIsrIsComplete`

- reinvoke a child's parent ISR function when the child member returns `kIsrIsNotComplete`

Using kIsrIsComplete

An ISR returning `kIsrIsComplete` starts the dispatching process back toward the root. In the current example, assume that interrupt set C has its dispatching modifier option set to reinvoke the parent when `kIsrIsComplete` is returned. When the traversal toward the root encounters the `InterruptMemberNumber` 3 of member set C, parent set member B of `InterruptMemberNumber` 2 has its ISR reinvoked. This ISR might then, for example, return an `InterruptMemberNumber` value of 2, which would invoke the ISR of member set C, Int`erruptMemberNumber` value 2. This ISR would service its device and returns `kIsrIsComplete`. Since no higher interrupt set has any dispatching modifier options, the dispatching process will arrive at the root and be finished.

In this way, the `kIsrIsComplete` dispatching option is typically used to give a parent member a chance to service additional children without having to reenter the dispatching process.

Using kIsrIsNotComplete

An ISR returning `kIsrIsNotComplete` produces slightly more complex behavior. An ISR returns `kIsrIsNotComplete` only when its device was not the device requesting service. Even though a leaf ISR was invoked, the interrupt request is still outstanding and the ISR for the requesting device must be found. If the member set containing the ISR just invoked has no dispatching modifying options, then the next interrupt member in the set will have its ISR invoked. In the current example, the ISR of IST member set D, `InterruptMemberNumber` 2, would be invoked. Assuming that this ISR serviced its device and returned `kIsrIsComplete`, dispatching would be complete since no higher interrupt set had any dispatching modifier options set.

If the ISR of IST member set D, `InterruptMemberNumber` 2, also returned `kIsrIsNotComplete`, however, the ISR of the next interrupt member in the parent set would be invoked. In the example, `InterruptMemberNumber` 3 of member set C is already the last member in set C, so this set is skipped and the next higher set is examined (in this case, set B). Set B is found to have higher members, resulting in the ISR of member set B, `InterruptMemberNumber` 3, being invoked. Assuming that this ISR serviced its device and returned `kIsrIsComplete`, dispatching would be finished.

The behavior just described is a classic left-branch recursive tree walk. It is employed when no means exist for directly identifying exactly which device is requesting service. Devices must be polled, by invoking their ISRs, to find and service the requesting device.

While this behavior will correctly poll for the requesting device, it is sometimes inappropriate to poll devices in the order that they appear in the member set. In the example, assume that interrupt set B has its dispatching modifier option set to reinvoke the parent ISR if `kIsrIsNotComplete` is returned. In the example just cited, when the traversal toward the root encounters `InterruptMemberNumber` 2 of member set B, the parent set member A, `InterruptMemberNumber` 1, has its ISR reinvoked. This ISR could then return `InterruptMemberNumber` 4 to invoke member set B, `InterruptMemberNumber` 4. In this way, `kIsrIsNotComplete` should be used when the priority of devices is not the same as the order in which devices appear in their member sets.

Interrupt Priority

Note that there is no explicit prioritization scheme reflected in this process, but that implied prioritization does take place. The fact that tree transversal proceeds from the root member toward leaf members gives members closer to the root a higher priority. Hence, the hierarchical structure of the IST determines the system's fixed interrupt priority structure. Conversely, a transversal ISR is free to use any algorithm to decide which child member's ISR should be invoked—for example, an anti-starvation algorithm or a priority based on the value of `InterruptMemberNumber`. Whatever method is used, transversal ISRs provide the dynamic aspect of system's interrupt priority structure. Implementing the IST structure and ISR usage sets the implied prioritization of all interrupts.

The Mac OS does set priority for hardware interrupts, separately from the software interrupt dispatch mechanism. It is therefore possible for hardware interrupt priority to cause an ISR to be interrupts.

Interrupt Source Tree Construction

The Mac OS startup process automatically performs the initial construction and maintenance of the IST for all built-in I/O ASICs, PCI expansion cards, and PCI-to-PCI bridges that use the default PCI bridge IST extensions.

Note
Expansion card developers normally have no need to construct the IST but may need to extend it as described in "Explicit IST Extension" (page 394). The following description of the initial construction process is included for completeness. ◆

The interrupt tree is constructed by creating new sets of child members under existing child members, which thus become parent members. The preexisting root member is used as the parent member for the first layer of the tree. As each new child member is created, a null ISR is installed and its IER and IDR routines are inherited from the parent. If built-in interrupt controller hardware can enable and disable interrupts for each of the interrupt members in the new interrupt set, IERs and IDRs tailored to each interrupt member are installed. When a child member becomes a parent member, a transversal ISR is installed on top of the null ISR for dispatching its child members. This process is repeated for as many layers and IST members as required. Typically, the default IST originally created services all the fixed hardware devices and slots on the Power Macintosh main logic board.

Having child members inherit their parents' IERs and IDRs allows devices that don't have hardware enabling and disabling support to still use IER and IDR functions. Invoking an IER or IDR for such a device will transparently invoke the parent member's IER or IDR. At some point up the interrupt tree, main logic board hardware will physically enable or disable interrupts intended for the device.

IMPORTANT
Default enablers, disablers, and transversal ISRs for all Macintosh built-in I/O devices are provided and installed by Apple I/O family experts. Drivers that use the family expert APIs are more portable and are more likely to be compatible with future Apple products. ▲

▲ **WARNING**
The Apple built-in handlers can be overridden by other software. However, built-in interrupt enablers, disablers, and transversal ISRs are very specific to the hardware platform. Detailed knowledge of the built-in interrupt controller hardware is required to successfully override one. ▲

Interrupts and the Name Registry

Once the IST is constructed and initialized, drivers need a mechanism to find the IST member that represents the interrupt source the driver is controlling. This is done through the Name Registry discussed in Chapter 10. As explained in "Initialization and Finalization Routines" (page 212), a driver's initialization call contains a `RegEntryID` value that refers to the set of Name Registry properties for the device the driver controls. Besides the standard set of PCI properties, a number of Apple-specific properties are included, as shown in Table 10-1 (page 322). The Apple property used for interrupts is `driver-ist`, which contains an array of interrupt sources logically associated with a device.

Each `driver-ist` property is stored as type `ISTProperty`, which is an array of three `InterruptSetMember` values (see "Basic Data Types" (page 396)), and conforms to the following rules:

- The first `InterruptSetMember` value contains the interrupt member for the device's controller chip or hardware interrupt source—for example, a serial controller chip or a card in an expansion slot. This interrupt member must always be defined for hardware that is capable of requesting hardware interrupts.
- If the device is capable of generating direct memory access (DMA) output interrupts, the second `InterruptSetMember` value contains the interrupt member for the interrupt source of the device's DMA output interrupts. Otherwise, it contains null values.
- If the device is capable of generating DMA input interrupts, the third `InterruptSetMember` value contains the interrupt member for the interrupt source of the device's DMA input interrupts. Otherwise, it contains null values.
- If the device generates both DMA input and output interrupts with the same interrupt source, the second `InterruptSetMember` value contains the interrupt member for both DMA input and output interrupts. In this case, the third `InterruptSetMember` contains null values.

Note that grouping these interrupt members in one `driver-ist` property is purely a logically grouping. Any one of the three interrupt members can be located anywhere within the IST hierarchy.

Extending the Interrupt Source Tree

This section discusses the ways that the IST can grow to accommodate PCI devices and bridges.

Automatic IST Extension

The construction process described in "Interrupt Source Tree Construction" (page 389) builds an IST for all devices that are connected directly to the main logic board's PCI bus. This includes all devices on the Power Macintosh main logic board plus expansion slots that are populated with single-function expansion cards. However, additional devices may exist that are indirectly connected to the main logic board's PCI bus by means of PCI-to-PCI bridges. Examples of such devices are PCI-to-PCI expansion chassis cards and multifunction expansion cards that use controller chips with built-in PCI interfaces.

A single-function device that is plugged into a main logic board slot will always have a pre-built IST member available because the slot is always present and accounted for when constructing the IST. Multifunction devices, based on PCI-to-PCI bridge devices, aren't treated so simply. While the pre-built IST member for the slot is still available for use by the multifunction device, the number of devices on the other side of the PCI-to-PCI bridge is unknown and must be accounted for.

Therefore, Mac OS dynamically extends the IST and the Name Registry during system initialization for all PCI-to-PCI bridges and for all devices behind them. Each PCI-to-PCI bridge and functional device gets its own Name Registry entry and IST member. This makes each PCI-to-PCI bridge and functional device appear separately in the Name Registry and IST regardless of how many devices are physically bundled together on the same expansion card. This is convenient for expansion cards that contains more than one copy of a controller chip (for example a 4-port Ethernet card). The driver developer needs only develop a driver that knows how to control a single controller chip or port; Mac OS will automatically create an instance of the driver for each device that matches the driver. While the driver developer can choose to override the default mechanism, using this service can greatly decrease the complexity of some drivers.

Automatic IST Extension Operation

The nature of the PCI-to-PCI bridge devices available on the market today imposes some limitations on automatic IST extensions. While today's PCI-to-PCI bridge devices transparently handle the addressing aspects of PCI buses, they do not do the same for interrupt request signals. Also, there is no current standard among card vendors for providing hardware registers that indicate which device is requesting service. Hence, card vendors often simply wire the interrupt request signals from all devices together into a single signal and feed that directly to the main logic board's slot. The IST that is constructed for the main logic board can tell that something wants service on the multifunction expansion card, but it cannot tell exactly which device. To accommodate this "lowest common denominator" behavior, the IST extensions from the slot IST member uses dispatching modification options to poll the extended IST members, as described in "InterruptHandler" (page 397).

When polling is used, certain actions must be observed by the ISRs, IERs, and IDRs attached to the extended IST members. Each PCI-to-PCI bridge's IST member has a special bridge dispatching ISR installed. This transversal ISR handles all the devices requesting interrupt service during a single IST transversal. Once all of the device's ISRs return `kIsrIsNotComplete`, the transversal ISR returns `kIsrIsComplete` to the dispatcher to indicate that interrupt processing is complete. The transversal ISR also implements a simple fairness algorithm that keeps any one device from dominating the interrupt service requests. It makes sure that the same device isn't serviced twice in a row (unless only one device is requesting service), regardless of the number of IST transversals.

In addition, separate software flags are maintained for each extended IST member to enable and disable interrupt servicing. Invoking an extended IST member's IDR and IER functions has two implicit effects. First, invoking the IDR only prevents the extended IST member's ISR from being invoked; it does not disable the device's ability to request an interrupt. It is the responsibility of the driver to disable interrupt requests from the actual device. Second, invoking the IER not only allows the extended IST member's ISR to be invoked; it also traverses the IST back to the main logic board's slot IST member, invoking the IER of each IST member encountered. Thus, a driver needs only invoke its device's IER to allow interrupt requests through the IST.

Explicit IST Extension

By the time the PCI devices built into the Macintosh system are initialized, an IST has been constructed and populated with nodes for every interrupt source within the system, including all PCI expansion cards and PCI-to-PCI bridges that use the default PCI bridge IST extensions.

However, PCI expansion devices that cannot use the default PCI bridge IST extensions or that have special requirements will not automatically receive nodes in the IST. Examples of such devices are multifunction cards with non-PCI controller devices and PCI-to-NuBus expansion chassis. Because these devices still represent additions to the system hardware, the third-party driver writer needs to provide software that extends both the Name Registry and the Apple-provided IST.

Note
PCI-to-NuBus expansion bus cards are a special case. NuBus devices are controlled by 68K drivers and so require the Macintosh facilities normally provided for NuBus devices. The interrupt handler for the PCI-to-NuBus bridge must use or provide Slot Manager dispatching and interrupt registration for NuBus device drivers. The initialization of a PCI-to-NuBus bridge does not need to extend the Registry or the IST. ◆

If you are extending the system by means of PCI bus slots or a multifunction device, the work to be done includes several basic steps:

- When the device initialization code is first invoked, it will be passed the `RegEntryID` value of the Registry node that represents the PCI expansion slot that the device occupies. Use the `RegistryPropertyGet` function to get the `driver-ist` property for the PCI expansion slot, which will have the `InterruptSetMember` value for the slot's interrupts.
- Pay particular attention to the fact that the parent (or bridge or multifunction) initialization code must be marked as initialize and open upon discovery. This is a requirement because extension devices must be available in the Name Registry before family experts are run. If this requirement is not met, extension devices may not be made available to the system because their child devices will not be found. Initialize and open upon discovery is described in "Driver Run-Time Structure" (page 201).
- Use the `GetInterruptFunctions` function with the slot's `InterruptSetMember` value to get the default IDR registered with the parent member. Call the IDR

to disable the parent member's interrupt propagation. This keeps spurious interrupts from occurring before the IST extension is complete.

- The device initialization code must extend the IST. Use the `CreateInterruptSet` function to create a new interrupt set with the slot's `InterruptSetMember` value as the parent member. Make the interrupt set size the same as the number of new PCI bus slots or the number of functions (in a multifunction device).
- Register a transversal ISR with the parent member, using the slot's `InterruptSetMember` value. When invoked, this transversal ISR should further route the slot interrupt to one of the interrupt members in the newly created interrupt set.
- If the device's interrupt controller hardware can enable and disable interrupts for each of the interrupt members in the new interrupt set, register tailored IERs and IDRs with each of the interrupt members. Otherwise, the IER and IDR that the interrupt members inherited from the parent member will moderate interrupts transparently to the caller.
- For each additional device or function, a node must also be added to the Name Registry. Adding nodes to the Registry is described in "Name Creation and Deletion" (page 294).
- Each new child entry in the Registry requires a complete set of properties to allow the device to be located by its family experts. A complete set of properties is the set of properties described by and installed by Open Firmware. For details, see the Open Firmware standard and Table 10-1 (page 322).
- In addition to the Open Firmware requirements, each new child entry in the Registry must also have a `driver-ist` property installed. This lets subsequent drivers that want to register an ISR with one of the newly created interrupt members find the correct `InterruptSetMember` value.
- Create properties using the rules described in the previous section and in "Property Management" (page 311). For each new child entry in the Registry, create a `driver-ist` property with the corresponding new interrupt members that were used to extend the IST.
- Call the IDR for each of the newly created interrupt members to keep spurious interrupts from occurring.
- Call the IER for the parent member to enable interrupts for the system extension as a whole.

Note
There will always be at least one new interrupt member created for each new child entry in the Name Registry. However, keep in mind that the `driver-ist` property is a logical grouping of interrupt members for a device or function. Because of this grouping, you might end up creating more interrupt members than child entries in the Registry. ◆

Native drivers can now be loaded against any of the new devices, as created by the extension to the IST and the Name Registry, just like other native drivers.

Basic Data Types

This section defines some data types and values that are fundamental to interrupt management.

```
typedef KernelID        InterruptSetID;
typedef long            InterruptMemberNumber;

enum {
kReturnToParentWhenComplete = 0x00000001,
kReturnToParentWhenNotComplete = 0x00000002
};

typedef struct InterruptSetMember {
    InterruptSetID                  set;
    InterruptMemberNumber           member;
} InterruptSetMember;

enum{
    kISTChipInterruptSource      = 0,
    kISTOutputDMAInterruptSource = 1,
    kISTInputDMAInterruptSource  = 2,
    kISTPropertyMemberCount      = 3
};

typedef InterruptSetMember ISTProperty[ kISTPropertyMemberCount ];

#define kISTPropertyName "driver-ist"
```

```
typedef long InterruptReturnValue;
enum
{
    kFirstMemberNumber          =   1,
    kIsrIsComplete              =   0
    kIsrIsNotComplete           = -1,
    kMemberNumberParent         = -2,
};

typedef Boolean InterruptSourceState;
enum
{
    kSourceWasEnabled           = true,
    kSourceWasDisabled          = false
};
```

Control Routines

This section describes three interrupt control routines, `InterruptHandler`, `InterruptEnabler`, and `InterruptDisabler`. Their use by native drivers is described in "Hardware Interrupt Mechanisms" (page 181). See also the sample code in Listing 11-3 (page 413).

InterruptHandler

```
InterruptMemberNumber InterruptHandler (
                     InterruptSetMember ISTmember,
                     void * refCon,
                     UInt32 theIntCount);
```

`--> ISTmember`	Member set ID of the IST member requesting service.
`--> refCon`	32-bit reference constant registered with the IST member.
`--> theIntCount`	Count of the number of interrupts processed, including the current one.

DESCRIPTION

When an ISR is invoked, `member` contains the ID of the IST member that is the currently interrupting source. Since an ISR can be registered with multiple IST members, the `member` parameter allows a single ISR to distinguish multiple interrupt sources. `RefCon` contains the reference constant that was installed along with the ISR.

If the ISR returns a positive number, the dispatcher uses that number to identify which child member should be invoked next.

If the ISR returns `kIsrIsComplete`, the interrupt dispatcher stops any further traversal of the IST and treats the interrupt request as serviced. See "Interrupt Dispatching" (page 386) for additional information about the interrupt service process.

IMPORTANT

Since an ISR can be invoked when the device the ISR services is not requesting service, an ISR must be able to detect this situation and return `kIsrIsNotComplete` to the dispatcher. This lets the dispatcher continue looking for the actual ISR that will service the interrupt request. ▲

The `theIntCount` parameter can be used by transversal interrupt handlers to determine if they have been reinvoked by the dispatcher. On each new interrupt tree transversal, this value is unique. This means that `theIntCount` will be a different value the first time a transversal ISR is invoked. However, if the transversal ISR is reinvoked during the same transversal process, the `theIntCount` value will be the same as the first time it was invoked. By saving the value of `theIntCount` during the previous tree traversal and verifying that the current value is the same, a transversal ISR can tell when it is being reinvoked.

Note that the `theIntCount` value will never be equal to `0`. On ISR installation, the ISR's saved copy of `theIntCount` should be initialized to `0` so that the first invocations of the ISR can behave properly.

IMPORTANT

The actual value of `theIntCount` shouldn't interpreted in any way. How this value is computed may change in the future. The only valid interpretation of `theIntCount` is that it is unique for each interrupt tree transversal process and that it will never be 0. ▲

InterruptEnabler

```
void InterruptEnabler (
                      InterruptSetMember ISTmember,
                      void * refCon);
```

`--> ISTmember`	Member set ID of the IST member requesting service.
`<--> refCon`	32-bit reference constant registered with the IST member.

DESCRIPTION

Invoking `InterruptEnabler` reenables the interrupt member's ability to propagate interrupts to Mac OS.

Note
Apple-defined enabler functions do not use the passed values of `refCon` and should therefore be passed `nil`. The `refCon` value lets user-defined enabler functions receive a reference constant of the programmer's choice.

InterruptDisabler

```
InterruptSourceState InterruptDisabler(
                      InterruptSetMember ISTmember,
                      void * refCon);
```

`--> ISTmember`	Member set ID of the IST member requesting service.
`<--> refCon`	32-bit reference constant registered with the IST member.

DESCRIPTION

Invoking `InterruptDisabler` disables the interrupt member's ability to propagate interrupts to Mac OS. This routine returns the member's ability to propagate interrupts as it was before the routine was invoked. A returned value of `kSourceWasEnabled` means that the interrupt member's propagation state was enabled; a returned value of `kSourceWasDisabled` means it was disabled.

Note
Apple-defined enabler functions do not use the passed values of `refCon` and should therefore be passed `nil`. The `refCon` value lets user-defined enabler functions receive a reference constant of the programmer's choice. ◆

Control Routine Installation and Examination

To install an interrupt handler, use `InstallInterruptFunctions`. This routine replaces the earlier Slot Manager routine `SIntInstall`. After an ISR has been installed, `GetInterruptFunctions` lets you examine it.

IMPORTANT
ISR functions are never explicitly removed. To deregister an ISR, reinstall the ISR function that was obtained by means of the `GetInterruptFunctions` routine before the ISR was originally installed. Then call the IST disabler function to keep any further interrupts from requesting service. ▲

The declarations for the interrupt handler, enabler, and disabler are the following:

```
typedef InterruptMemberNumber   (*InterruptHandler)
                                (InterruptSetMember ISTmember,
                                void *              refCon,
                                UInt32              theIntCount);

typedef void            (*InterruptEnabler)
                        (InterruptSetMember member,
                         void *             refCon);

typedef InterruptSourceState    (*InterruptDisabler)
                                (InterruptSetMember member,
                                void *              refCon);
```

The interrupt set ID and interrupt member number values are available as `driver-ist` properties associated with each device entry in the Name Registry. Hardware, secondary, and software interrupt mechanisms are described in "Interrupt Management" (page 381).

InstallInterruptFunctions

The `InstallInterruptFunctions` function installs interrupt service routines in an interrupt member.

```
OSStatus InstallInterruptFunctions (
                    InterruptSetID setID,
                    InterruptMemberNumber member,
                    void *refCon,
                    InterruptHandler handlerFunction,
                    InterruptEnabler enableFunction,
                    InterruptDisabler disableFunction);
```

`--> setID`	Interrupt set ID of the IST member to be installed.
`--> member`	Set member number of the IST member to be installed.
`<-- refCon`	32-bit reference constant to be registered with the IST member.
`--> handlerFunction`	Pointer to interrupt service routine (ISR).
`--> enableFunction`	Pointer to interrupt enabler routine (IER).
`--> disableFunction`	Pointer to interrupt disabler routine (IDR).

DESCRIPTION

Given the ID of an interrupt set in the interrupt tree and the number of a member in that set, `InstallInterruptFunctions` installs the designated interrupt handler, enabler, and disabler routines. Interrupt sets and the interrupt tree are discussed in “Interrupt Management” (page 381).

Parameter `refCon` can be any 32-bit value. Mac OS does not use it; it is merely stored and passed to each invocation of the most recently installed ISR routine. Placing `nil` in a `handlerFunction`, `enableFunction`, or `disableFunction` parameter will not install a new routine—it will leave the current routine installed.

`InstallInterruptFunctions` returns `noErr` if the installation succeeded.

EXECUTION CONTEXT

`InstallInterruptFunctions` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

GetInterruptFunctions

```
OSStatus GetInterruptFunctions (
                    InterruptSetID setID,
                    InterruptMemberNumber member,
                    void **refCon,
                    InterruptHandler *handlerFunction,
                    InterruptEnabler *enableFunction,
                    InterruptDisabler *disableFunction);
```

`--> setID`	Interrupt set ID of the IST member.
`--> member`	Member set ID of the IST member.
`<-- refCon`	Pointer to returned reference constant.
`<-- handlerFunction`	Pointer to returned interrupt handler.
`<-- enableFunction`	Pointer to returned interrupt enabler function.
`<-- disableFunction`	Pointer to returned interrupt disabler function.

DESCRIPTION

The `GetInterruptFunctions` function fetches interrupt control routines installed in an interrupt member. The caller passes the member set ID and the set member number in `setID` and `member` to uniquely identify the interrupt member in the tree.

Upon successful completion, `GetInterruptFunctions` returns the reference constant, the ISR, the IER, and the IDR to the caller.

EXECUTION CONTEXT

`GetInterruptFunctions` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

Interrupt Set Creation and Options

The routines described in this section deal with interrupt sets. `CreateInterruptSet` extends an IST by creating a new interrupt set. `GetInterruptSetOptions` helps an expert determine how the interrupt dispatcher will handle an interrupt set, and `ChangeInterruptSetOptions` helps it change that behavior.

IMPORTANT

The Mac OS IST for PCI cards is initialized and activated by Apple software. Third-party I/O software needs only to update member functions as necessary to support PCI cards. Extending the IST is required only for multifunction cards and bridges that don't use the default PCI bridge IST extensions. ▲

CreateInterruptSet

```
OSStatus CreateInterruptSet (
                 InterruptSetID parentSet,
                 InterruptMemberNumber parentMember,
                 InterruptMemberNumber setSize,
                 InterruptSetID *setID,
                 InterruptSetOptions options);
```

`-->` `parentSet` Member set ID.

`-->` `parentMember`
Set member number.

`-->` `setSize` Number of child members to create.

`<-->` `setID` Interrupt set ID.

`-->` `options` Options:
`kReturnToParentWhenComplete = 0x00000001`
`kReturnToParentWhenNotComplete = 0x00000002`

DESCRIPTION

The `CreateInterruptSet` function extends an IST. When calling it, pass the member set ID and the set member number in `parentSet` and `parentMember` to uniquely identify which leaf member is to become the parent member. Pass the number of child members to create in `setSize`. Pass a pointer to a variable of type `InterruptSetID` in `setID`. `CreateInterruptSet` returns `noErr` if the creation process succeeded, and the variable pointed to by `setID` contains the member set ID of the new set's child members.

The `options` parameter operates in these ways to modify the default interrupt dispatching behavior:

- Option `kReturnToParentWhenComplete` modifies the behavior for successful interrupt completion. Any time a child in a set with this option returns `kIsrIsComplete`, the dispatcher reinvokes the parent's transversal ISR. A parent can thus reevaluate its children's interrupt requests and can have another child serviced immediately instead of having to traverse the entire interrupt tree again.
- Option `kReturnToParentWhenNotComplete` modifies the behavior for unsuccessful interrupt completion. Any time a child in a set with this option returns `kIsrIsNotComplete`, the dispatcher reinvokes the parent's transversal ISR. The parent can then invoke another child to try to service the interrupt request. This process is repeated until one of the children members returns `kIsrIsComplete` or the parent returns `kIsrIsNotComplete`. In the latter case, the dispatcher continues traversing the tree between the parent and the root, looking for an ISR to satisfy the interrupt request. If the root is reached, the interrupt request is treated as spurious.

- If no options are set, the dispatcher traverses the tree toward the root, looking for an IST member's interrupt set that has options set, until it arrives at the root.

EXECUTION CONTEXT

`CreateInterruptSet` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter
`memFullErr`	–108	Not enough room in heap

GetInterruptSetOptions

```
OSStatus GetInterruptSetOptions(
                    InterruptSetID setID,
                    InterruptSetOptions *options);
```

`--> setID`	Interrupt set ID of the interrupt set.
`<-- options`	Current dispatching options.

DESCRIPTION

`GetInterruptSetOptions` returns in `options` the dispatching behavior options for the interrupt set identified by `setID`.

EXECUTION CONTEXT

`GetInterruptSetOptions` may be called only from task level, not from software or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

`ChangeInterruptSetOptions`

```
OSStatus ChangeInterruptSetOptions(
                    InterruptSetID setID,
                    InterruptSetOptions *options);
```

`--> setID`	Interrupt set ID of the interrupt set.
`--> options`	New dispatching options.

DESCRIPTION

`ChangeInterruptSetOptions` lets an expert change the behavior of the interrupt dispatcher for a specified interrupt set. The default behavior for most set members is to return to the root. For a multifunction PCI card the desired behavior might be to return to the parent, so the interrupt dispatcher can revisit all set members to determine whether all interrupts have been serviced or there is another to handle.

EXECUTION CONTEXT

`ChangeInterruptSetOptions` may be called only from task level, not from secondary or hardware interrupt level.

RESULT CODES

`noErr`	0	No error

Software Interrupts

The Driver Services Library provides several routines to create, run, and remove software interrupts.

CurrentTaskID

```
TaskID CurrentTaskID (void);
```

DESCRIPTION

`CurrentTaskID` returns the ID number of the currently running task. This routine can be called only from the noninterrupt execution level.

EXECUTION CONTEXT

`CurrentTaskID` may be called only from task level, not from secondary or hardware interrupt level.

CreateSoftwareInterrupt

```
OSStatus CreateSoftwareInterrupt(
                    SoftwareInterruptHandlerhandler,
                    TaskID task,
                    const void *p1,
                    Boolean persistent,
                    SoftwareInterruptID *softwareInterrupt)
```

`--> handler`	Handler for the new software interrupt.
`--> task`	Task ID.
`--> p1`	First parameter to be passed to the handler.
`--> persistent`	Indicates whether the ID of the software interrupt should be deleted when it is activated or should persist until deleted by `DeleteSoftwareInterrupt`.
`--> theSoftwareInterrupt`	Software interrupt ID.

DESCRIPTION

`CreateSoftwareInterrupt` creates a software interrupt for a specified task. It can be called either from noninterrupt or secondary execution level.

Persistent software interrupts may be sent multiple times but only once per activation; that is, the software interrupt must run before it can be sent again.

EXECUTION CONTEXT

`CreateSoftwareInterrupt` may be called from task level or secondary interrupt level but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Bad parameter

SendSoftwareInterrupt

```
OSStatus SendSoftwareInterrupt(
                      SoftwareInterruptID softwareInterrupt,
                      const void *p2);
```

`--> softwareInterrupt`
Software interrupt ID.

`--> p2` First parameter to be passed to the handler.

DESCRIPTION

`SendSoftwareInterrupt` causes a task to run a software interrupt. It can be called from any execution level and acts as an asynchronous function.

IMPORTANT

Currently, `SendSoftwareInterrupt` calls the user back at the same execution level. In future versions of Mac OS it can be used to force execution of code that can't be called at interrupt level. ▲

EXECUTION CONTEXT

`SendSoftwareInterrupt` may be called from task level or secondary interrupt level but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Queue element not found
`paramErr`	–50	Bad parameter

DeleteSoftwareInterrupt

```
OSStatus DeleteSoftwareInterrupt (SoftwareInterruptID softwareInterrupt)
```

`--> softwareInterrupt`
Software interrupt ID.

DESCRIPTION

`DeleteSoftwareInterrupt` removes a software interrupt.

EXECUTION CONTEXT

`DeleteSoftwareInterrupt` may be called from task level or secondary interrupt level but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Queue element not found
`paramErr`	–50	Bad parameter

Secondary Interrupt Handlers

Secondary interrupt handlers are the primary synchronization mechanism that a driver and its hardware interrupt handlers may use. Secondary interrupt handlers must conform to the interrupt execution environment rules, including absence of page faults, severe restrictions on using system services, and so on. For further information, see "Device Driver Execution Contexts" (page 346).

The special characteristic of secondary interrupt handlers that makes them useful is that the operating system guarantees that at most one secondary handler is active at any time. This means that if you have a data structure that requires complex update operations and each of the operations uses secondary interrupt handlers to access or update the data structure, then all access to the data structure will be atomic even though hardware interrupts are enabled during the access.

The DSL provides timers that can run secondary interrupt handlers when they expire. See "Interrupt Timers" (page 422).

Note
Although interrupts are accepted during the execution of secondary interrupt handlers, no noninterrupt level execution can take place. This can lead to severely degraded system responsiveness. Use the secondary interrupt facility only when necessary. ◆

Secondary interrupt handlers have the form shown in the next section.

IMPORTANT
Secondary interrupts can't be used on the page fault path with Mac OS prior to Mac OS 8.5. ▲

SecondaryInterruptHandlerProc2

```
typedef OSStatus (*SecondaryInterruptHandlerProc2)
                    (void *p1,
                     void *p2);
```

`--> p1`	First parameter.
`--> p2`	Second parameter.

DESCRIPTION

The secondary interrupt handler you write must have the interface shown above, with two parameters. You must specify the values of the two parameters at the time you queue the handler. For queuing information, see the next section.

RESULT CODE REQUIRED

`noErr`	0	No error
`Err`	–1	Routine failed

Queuing Secondary Interrupt Handlers

Secondary interrupt handlers are usually queued during the processing of a hardware interrupt. A secondary interrupt handler's execution will be deferred until processing is about to move back to noninterrupt level. You may, however, queue secondary interrupt handlers from secondary interrupt level. In this case, the queued handler will be run after all other such queued handlers, including the current handler, have finished.

Secondary interrupt handlers that are queued from hardware interrupt handlers consume memory resources from the time they are queued until the time they finish execution. They do this regardless of the execution context (see "Device Driver Execution Contexts" (page 346)). You should make every attempt to limit the number of simultaneously queued secondary interrupt handlers because the memory resources available to them are limited. Only one kind of secondary interrupt handler, that with two parameters, may be queued.

QueueSecondaryInterruptHandler

```
OSStatus QueueSecondaryInterruptHandler(
                     SecondaryInterruptHandler2 handler,
                     ExceptionHandler exceptionHandler,
                     const void *p1,
                     const void *p2);
```

`--> handler` The handler to be queued.

--> `exceptionHandler`	Exception handler (not currently implemented, pass nil).
--> `p1`	First handler parameter.
--> `p2`	Second handler parameter.

DESCRIPTION

`QueueSecondaryInterruptHandler` queues the secondary interrupt handler indicated by `handler`. Future versions of Mac OS may allow an exception handler to be associated with the interrupt handler; the `exceptionHandler` parameter is currently ignored, always pass nil.

EXECUTION CONTEXT

`QueueSecondaryInterruptHandler` may be called from task level, secondary interrupt level, or hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`qErr`	–1	Queue element not found

Calling Secondary Interrupt Handlers

Secondary interrupt handlers can be called synchronously by the function `CallSecondaryInterruptHandler2`. This service may be used from either noninterrupt level or secondary interrupt level but not from hardware interrupt level.

CallSecondaryInterruptHandler2

```
OSStatus CallSecondaryInterruptHandler2(
                     SecondaryInterruptHandlerProc2 handler,
                     ExceptionHandler exceptionHandler,
                     const void *p1,
                     const void *p2);
```

`-->` `handler`	The handler to be queued.
`-->` `exceptionHandler`	Exception handler (not currently implemented).
`-->` `p1`	First handler parameter.
`-->` `p2`	Second handler parameter.

DESCRIPTION

`CallSecondaryInterruptHandler2` calls the secondary interrupt handler indicated by `handler`. The secondary interrupt handler is invoked immediately; it is not queued.

EXECUTION CONTEXT

`CallSecondaryInterruptHandler2` may be called from task level or secondary interrupt level, but not from hardware interrupt level.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Call failed

Interrupt Code Example

The code sample in Listing 11-3 shows a typical interrupt registration process during driver initialization.

Listing 11-3 Interrupt registration

```
#include <Devices.h>
#include <Interrupts.h>
#include <NameRegistry.h>

// useful global data within my driver
```

```
DriverRefNum                myDriverRefNum;
RegEntryID                  myRegEntryID;
InterruptSetMember          myISTMember;
void *                      theDefaultRefCon;
InterruptHandler            theDefaultHandlerFunction;
InterruptEnabler            theDefaultEnableFunction;
InterruptDisabler           theDefaultDisableFunction;

// the ISR function to be registered

InterruptMemberNumber
myISRHandler(       InterruptSetMember          member,
                    void *                      refCon,
                    UInt32                      theIntCount)
    {

    Boolean myDeviceWantsService( void );
    void serviceMyDevice( void );

    // see if your device was the one that requested an interrupt
    if( myDeviceWantsService() == false )
        return kIsrIsNotComplete

    // do what ever is required to service your hardware here
    serviceMyDevice();

    // tell the system that this interrupt has been serviced
    return kIsrIsComplete;
    }

// the main entry point for interrupt initialization

OSErr
DoInitializeCommand(        DriverRefNum        myRefNum,
                            RegEntryID          myRegID )
    {
    OSErr                           Status;
    RegPropertyValueSize            propertySize;
    ISTProperty                     theISTProperty;
```

```
// remember our RefNum and Registry Entry ID
myDriverRefNum = myRefNum;
myRegEntryID   = myRegID;

// get 'driver-ist' property from the Registry for my device
propertySize = sizeof( theISTProperty );

Status = RegistryPropertyGet(           &myRegEntryID,
                                        kISTPropertyName,
                                        theISTProperty,
                                        &propertySize );

// return if we got an error
if( Status != noErr )
    return Status;

// remember the first InterruptSetMember in the 'driver-ist'
// as the IST member that my driver is connected to
myISTMember.setID = theISTProperty[ kISTChipInterruptSource ].setID;
myISTMember.member = theISTProperty[ kISTChipInterruptSource ].member;

// get the default "enabler" function for my IST member
Status = GetInterruptFunctions(             myISTMember.setID,
                                            myISTMember.member,
                                            &theDefaultRefCon,
                                            &theDefaultHandlerFunction,
                                            &theDefaultEnableFunction,
                                            &theDefaultDisableFunction );

// return if we got an error
if( Status != noErr )
    return Status;

// register my ISR with my IST member. Don't register an
// "enabler" or "disabler" function since the IST member
// my driver is connected to is a Macintosh on-board device.
Status = InstallInterruptFunctions(             myISTMember.setID,
                                               myISTMember.member,
                                               0,
                                                        (InterruptHandler)myISRHandler,
                                                       (InterruptEnabler)0,
                                                       (InterruptDisabler)0 );

// return if we got an error
if( Status != noErr )
    return Status;

// make sure that interrupts are enabled for my IST member
theDefaultEnableFunction(           myISTMember,
                                   0 );

return Status;
}
```

Timing Services

The timing services that the Driver Services Library provides to device drivers allow the precise measurement of elapsed time as well as the execution of secondary interrupt handlers at desired times.

The accuracy of timer operations is quite good. However, certain limitations are inherent in the timing mechanisms. These are described below.

Time Base

Timer hardware within the system is clocked at a rate that is model dependent. This rate is called the **time base.** The timing services isolate software from the time base by representing all times in `AbsoluteTime` values, the units required by the timing services. You may use conversion routines to convert from `Nanoseconds` or `Duration` values into `AbsoluteTime` system units. This conversion can introduce errors, but errors are typically limited to one unit of the time base.

Representing the time base is difficult; the value is typically an irrational number. Mac OS solves this problem by returning a representation of the time base in fractional form—two 32-bit integer values, a numerator and denominator. If you multiply an `AbsoluteTime` value by the value of numerator and divide the result by the value of denominator, the result is nanoseconds.

When performing sensitive timing operations, it can be important to know the underlying time base. For example, if the time base is 10 milliseconds, there is little value in setting timers for 1 millisecond. You can determine the hardware time base by using `GetTimeBaseInfo`.

GetTimeBaseInfo

```
void GetTimeBaseInfo (
                    UInt32 *minAbsoluteTimeDelta,
                    UInt32 *theAbsoluteTimeToNanosecondNumerator,
                    UInt32 *theAbsoluteTimeToNanosecondDenominator,
                    UInt32 *theProcessorToAbsoluteTimeNumerator,
                    UInt32 *theProcessorToAbsoluteTimeDenominator);
```

`<-- minAbsoluteTimeDelta`
Minimum number of `AbsoluteTime` units between time changes.

`<-- theAbsoluteTimeToNanosecondNumerator`
Absolute to nanoseconds numerator.

`<-- theAbsoluteTimeToNanosecondDenominator`
Absolute to nanoseconds denominator.

`<-- theProcessorToAbsoluteTimeNumerator`
Processor time to absolute numerator.

`<-- theProcessorToAbsoluteTimeDenominator`
Processor time to absolute denominator.

DESCRIPTION

The `GetTimeBase` function returns information used to determine the current hardware time base in fractional form—two 32-bit integer values, a numerator and denominator. The `minAbsoluteTimeDelta` value is the minimum number of `AbsoluteTime` units that can change at any given time. If you multiply an `AbsoluteTime` value by the value of `theAbsoluteTimeToNanosecondNumerator` and divide the result by the value of `theAbsoluteTimeToNanosecondDenominator`, the result is nanoseconds.For example, if the Power Macintosh hardware changes the decrementer in quantities of 128, then the `minAbsoluteTimeDelta` value returned by `GetTimeBaseInfo` would be 128.

EXECUTION CONTEXT

`GetTimeBaseInfo` may be called from task level, secondary interrupt level, or hardware interrupt level.

Measuring Elapsed Time

Measurement of elapsed time is done by simply obtaining the time before and after the event to be timed. The difference of these two values indicates the elapsed time. Time, in this context, refers to a 64-bit `AbsoluteTime` count maintained by Mac OS. The count is set to 0 by the operating system during its initialization at system startup time. Conversion routines are provided in a shared library to convert from `AbsoluteTime` to 64-bit `Nanoseconds` or 32-bit `Duration` values.

Basic Time Types

Callers wishing to specify a time relative to the present use the type `Duration`:

```
typedef long Duration;
```

Values of type `Duration` are 32 bits long. They are interpreted in a manner consistent with the Time Manager—positive values are in units of milliseconds, negative values are in units of microseconds. Therefore the value 1500 is 1500 milliseconds or 1.5 seconds while the value –8000 is 8000 microseconds or 8 milliseconds. Notice that many values can be expressed in two different ways. For example, 1000 and –1000000 both represent exactly one second. When two representations have equal value, they may be used interchangeably; neither is preferred or inherently more accurate.

Values of type `Duration` may express times as short as 1 microsecond or as long as 24 days. However, two values of type `Duration` are reserved and have special meaning. The value `durationImmediate` specifies no duration. The value `durationForever`, the largest positive 32-bit value, specifies that many milliseconds, or a very long time from the present.

The Driver Services Library provides the following definitions for use with values of type `Duration`:

```
enum
{
    durationMicrosecond             = -1,
    durationMillisecond             = 1,
    durationSecond                  = 1000,
    durationMinute                  = 1000 * 60,
    durationHour                    = 1000 * 60 * 60,
    durationDay                     = 1000 * 60 * 60 * 24,
    durationForever                 = 0x7FFFFFFF,
    durationImmediate               = 0,
};
```

Another form for representing time is in `Nanoseconds`, the values of which are represented by unsigned 64-bit integers:

```
typedef struct Nanoseconds
{
    unsigned long   hi;
    unsigned long   lo;
} Nanoseconds;
```

A second data type, `AbsoluteTime`, is used to specify absolute times in system-defined units 64 bits long. As discussed in “Time Base” (page 416), the real duration of `AbsoluteTime` units must be calculated.

```
typedef struct AbsoluteTime
{
    unsigned long   hi;
    unsigned long   lo;
} AbsoluteTime;
```

Obtaining the Time

You can read the internal representation of time to which all timer services are referenced. This value starts at 0 during operating-system initialization and increases throughout the system’s lifetime.

UpTime

```
AbsoluteTime UpTime (void);
```

DESCRIPTION

`UpTime` returns the time since OS initialization in `AbsoluteTime` units.

EXECUTION CONTEXT

`UpTime` may be called from task level, secondary interrupt level, or hardware interrupt level.

Time Conversion Routines

The Driver Services Library provides the following conversion routines to convert between `Nanoseconds`, `Duration`, and `AbsoluteTime` units:

```
Nanoseconds AbsoluteToNanoseconds (AbsoluteTime absoluteTime);

Nanoseconds DurationToNanoseconds (Duration duration);

Duration AbsoluteToDuration (AbsoluteTime absoluteTime);

AbsoluteTime NanosecondsToAbsolute (Nanoseconds nanoseconds);

AbsoluteTime DurationToAbsolute (Duration duration);

Duration NanosecondsToDuration (Nanoseconds nanoseconds);

AbsoluteTime AddAbsoluteToAbsolute(
                    AbsoluteTime absoluteTime1,
                    AbsoluteTime absoluteTime2);

AbsoluteTime SubAbsoluteFromAbsolute(
                    AbsoluteTime leftAbsoluteTime,
                    AbsoluteTime rightAbsoluteTime);

AbsoluteTime AddNanosecondsToAbsolute(
                    Nanoseconds nanoseconds,
                    AbsoluteTime absoluteTime);

AbsoluteTime AddDurationToAbsolute(
                    Duration duration,
                    AbsoluteTime absoluteTime);

AbsoluteTime SubNanosecondsFromAbsolute(
                    Nanoseconds nanoseconds,
                    AbsoluteTime absoluteTime);

AbsoluteTime SubDurationFromAbsolute(
                    Duration duration,
                    AbsoluteTime absoluteTime);
```

```
Nanoseconds AbsoluteDeltaToNanoseconds(
                     AbsoluteTime leftAbsoluteTime,
                     AbsoluteTime rightAbsoluteTime);

Duration AbsoluteDeltaToDuration(
                     AbsoluteTime leftAbsoluteTime,
                     AbsoluteTime rightAbsoluteTime);
```

Note
The value of `rightAbsoluteTime` is usually larger than that of `leftAbsoluteTime`. However, if you subtract a `rightAbsoluteTime` value from a `leftAbsoluteTime` value, the result is 0, not a negative number. ◆

EXECUTION CONTEXT

The time conversion routines may be called from task level, secondary interrupt level, or hardware interrupt level.

Interrupt Timers

Interrupt timers allow you to specify that a secondary interrupt handler is to run when the timer expires. They are asynchronous in nature.

Setting Interrupt Timers

You can set an interrupt timer from any driver execution context. Each interrupt timer is identified by a timer ID:

```
typedef KernelID TimerID;
```

IMPORTANT
Interrupt timers consume memory resources from the time they are invoked until the time they expire or are canceled. They do this regardless of the execution context (see "Device Driver Execution Contexts" (page 346)). You should make every attempt to limit the number of interrupt timers because the memory resources available to them are limited. ▲

SetInterruptTimer

```
OSStatus SetInterruptTimer (
                    const AbsoluteTime *expirationTime,
                    SecondaryInterruptHandler2 handler,
                    void *p1,
                    TimerID *timer);
```

`--> expirationTime`	Time when the timer expires.
`--> handler`	Address of a secondary interrupt handler.
`--> p1`	First parameter to be passed to handler.
`<-- timer`	Timer ID.

DESCRIPTION

The parameter `expirationTime` is the current time plus the amount of time delay before calling the interrupt handler, expressed in `AbsoluteTime` units.

Parameter `handler` is the address of a secondary interrupt handler that is to be run when the specified time is reached.

Parameter `p1` is the value that is passed as the first parameter to the secondary interrupt handler when the timer expires. The value of the second parameter passed to the secondary interrupt handler is set to the current program counter at the time the timer expired.

Parameter `timer` is updated with the ID of the timer that is created. This ID may be used in conjunction with `CancelTimer`, described on (page 425).

IMPORTANT

If you use `SetInterruptTimer` in your code, you must provide a copy of System Enabler version 1.0.1 to Power Macintosh 9500 users who have Enabler version 1.0. If Enabler version 1.0.1 or later is already installed, the installer should not replace it. Only the Power Macintosh 9500 has a problem with `SetInterruptTimer`, and it occurs on only a few early units. Other Power Macintosh models are not affected. For further information, see the folder "New 9500 Enabler" in the PCI Device Driver Kit. ▲

EXECUTION CONTEXT

`SetInterruptTimer` may be called from task level, secondary interrupt level, or hardware interrupt level.

RETURN CODE

`noErr`	0	No error

DelayFor

```
OSStatus DelayFor (Duration expirationTime);
```

`--> expirationTime`
Amount of time to delay.

DESCRIPTION

`DelayFor` blocks execution for a given time. Parameter `expirationTime` is the amount of time to suspend execution, expressed as a positive number in milliseconds or as a negative number in microseconds.

EXECUTION CONTEXT

`DelayFor` may be called only from task level, not from secondary or hardware interrupt level.

RETURN CODES

`noErr`	0	No error
`Err`	–1	Routine failed

DelayForHardware

```
OSStatus DelayForHardware (AbsoluteTime absoluteTime);
```

--> `absoluteTime`
Amount of time to delay.

DESCRIPTION

`DelayForHardware` spins execution for a given time, so the computer does no useful work. Parameter `absoluteTime` is the amount of time to delay in processor-dependent units. You can call `NanosecondsToAbsolute` to obtain timing for the processor in your system. `DelayFor`Hardware may be called at any execution level.

EXECUTION CONTEXT

`DelayForHardware` may be called from task level, secondary interrupt level, or hardware interrupt level.

RETURN CODES

`noErr`	0	No error
`Err`	–1	Routine failed

Canceling Interrupt Timers

Currently running asynchronous timers can be canceled. When you attempt to cancel an asynchronous timer a race condition begins between your cancellation request and expiration of the timer. It is therefore possible that the timer will expire and that your cancellation attempt will fail even though the timer had not yet expired at the instant the cancellation attempt was made.

With current versions of the Mac OS, if a primary interrupt handler queues a secondary handler that is to cancel a timer by calling `CancelTimer`, and if the secondary handler queues another secondary handler, the operating system guarantees that the timer will either execute or be canceled before the other secondary handler runs.

CancelTimer

```
OSStatus CancelTimer (TimerID timer, AbsoluteTime *timeRemaining);
```

`--> timer`	Timer ID.
`<-- timeRemaining`	Time left on timer when it was canceled.

DESCRIPTION

`CancelTimer` cancels a timer previously created by `SetInterruptTimer`, described on (page 423). It returns in `timeRemaining` the amount of time that was left in the timer when it was canceled. It returns an error if the timer has either already expired or been canceled.

EXECUTION CONTEXT

`CancelTimer` may be called from task level, secondary interrupt level, or hardware interrupt level.

RETURN CODES

`noErr`	0	No error
`Err`	–1	Routine failed

Atomic Memory Operations

This section describes DSL functions that manipulate the contents of memory.

Byte Operations

The Driver Services Library provides several 32-, 16-, and 8-bit atomic memory operations for use by device drivers. These routines take logical address pointers and ensure that the operations are atomic with respect to all devices (for example, other processors and DMA engines) that participate in the coherency architecture of the Power Macintosh system.

IMPORTANT

Memory locations used by these operations must be long word aligned; if they are stored in a structure, you should use the compiler directive `#pragma options align=power`. ▲

```
Boolean
CompareAndSwap (long oldValue, long newValue, long *Value);

SInt32      IncrementAtomic     (SInt32 *value);
SInt32      DecrementAtomic     (SInt32 *value);

SInt32      AddAtomic           (SInt32 amount,        SInt32 *value);

UInt32      BitAndAtomic        (UInt32 mask,          UInt32 *value);
UInt32      BitOrAtomic         (UInt32 mask,          UInt32 *value);
UInt32      BitXorAtomic        (UInt32 mask,          UInt32 *value);

SInt16      IncrementAtomic16   (SInt16 *value);
SInt16      DecrementAtomic16   (SInt16 *value);
SInt16      AddAtomic16         (SInt32 amount,        SInt16 *value);
UInt16      BitAndAtomic16      (UInt32 mask,          UInt16 *value);
UInt16      BitOrAtomic16       (UInt32 mask,          UInt16 *value);
UInt16      BitXorAtomic16      (UInt32 mask,          UInt16 *value);

SInt8       IncrementAtomic8    (SInt8 *value);
SInt8       DecrementAtomic8    (SInt8 *value);
SInt8       AddAtomic8          (SInt32 amount,        SInt8 *value);
UInt8       BitAndAtomic8       (UInt32 mask,          UInt8 *value);
UInt8       BitOrAtomic8        (UInt32 mask,          UInt8 *value);
UInt8       BitXorAtomic8       (UInt32 mask,          UInt8 *value);
```

DESCRIPTION

The atomic routines perform various operations on the memory address specified by `value`:

- The `CompareAndSwap` routine compares the value at the specified address with `oldValue`. The value of `newValue` is written to the specified address only if `oldValue` and the value at the specified address are equal. `CompareAndSwap` returns `true` if `newValue` is written to the specified address; otherwise, it returns `false`. A `false` return value does not imply that `oldValue` and the value at the specified address are not equal; it only implies that `CompareAndSwap` did not write `newValue` to the specified address.
- `IncrementAtomic` increments the value by 1 and `DecrementAtomic` decrements it by 1. These functions return the value as it was before the change.

- `AddAtomic` adds the specified amount to the value at the specified address and returns the result.
- `BitAndAtomic` performs a logical `and` operation between the bits of the specified mask and the value at the specified address, returning the result. Similarly, `BitOrAtomic` performs a logical `OR` operation and `BitXorAtomic` performs a logical `XOR` operation.

EXECUTION CONTEXT

The atomic operation routines may be called from task level, secondary interrupt level, or hardware interrupt level.

Bit Operations

```
Boolean TestAndSet(
                    UInt32 bit
                    UInt8 *startAddress);

Boolean TestAndClear(
                    UInt32 bit
                    UInt8 *startAddress);
```

`--> bit`	The bit number in the range 0 through 7.
`--> startAddress`	The address of the byte in which the bit is located.

DESCRIPTION

`TestAndSet` sets and `TestAndClear` clears a single bit in a byte at a specified address. They return `true` if the bit was already set or cleared respectively and `false` otherwise.

EXECUTION CONTEXT

`TestAndSet` and `TestAndClear` may be called from task level, secondary interrupt level, or hardware interrupt level.

Queue Operations

The Driver Services Library provides the following I/O parameter block queue manipulation functions:

```
OSErr PBQueueCreate         (QHdrPtr *qHeader);
OSErr PBQueueInit           (QHdrPtr qHeader);
OSErr PBQueueDelete         (QHdrPtr qHeader);

void  PBEnqueue             (QElemPtr qElement,         QHdrPtr qHeader);
OSErr PBEnqueueLast         (QElemPtr qElement,         QHdrPtr qHeader);
OSErr PBDequeue             (QElemPtr qElement,         QHdrPtr qHeader);
OSErr PBDequeueFirst        (QHdrPtr qHeader,       QElemPtr
*theFirstqElem);
OSErr PBDequeueLast         (QHdrPtr qHeader,       QElemPtr
*theLastqElem);
```

DESCRIPTION

`PBQueueCreate` creates a new I/O parameter block queue. `PBQueueInit` initializes it and `PBQueueDelete` deletes it. `PBEnqueue` places the element pointed to by `qElement` next in the queue and `PBEnqueueLast` places it last. `PBDequeue` removes the next element in the queue. `PBDequeueFirst` removes the first element and `PBDequeueLast` removes the last element. For detailed information about the I/O parameter block queue, see *Inside Macintosh: OS Utilities.*

EXECUTION CONTEXT

The three queue routines, `PBQueueInit`, `PBQueueCreate`, and `PBQueueDelete`, may be called only from task level, not from software or hardware interrupt level.

The five queue element routines may be called from task level, secondary interrupt level, or hardware interrupt level.

RETURN CODES (QUEUE ROUTINES)

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap

RETURN CODES (ELEMENT ROUTINES)

`noErr`	0	No error
`qErr`	–1	Queue element not found

String Operations

The DSL provides a number of C and Pascal string manipulation functions that are available to drivers.

EXECUTION CONTEXT

All the string operation routines may be called from task level, secondary interrupt level, or hardware interrupt level.

StrCopy

```
StringPtr PStrCopy (StringPtr dst, ConstStr255Param src);

char *CStrCopy (char *dst, const char *src);
```

DESCRIPTION

The `PStrCopy` function copies the Pascal string from `src` to `dst`. `CStrCopy` copies characters up to and including the null character from `src` to `dst` C strings. These routines assume that the two strings do not overlap.

StrNCopy

```
StringPtr PStrNCopy (StringPtr dst, ConstStr255Param src, UInt32 max);

char *CStrNCopy (char *dst, const char *src, UInt32 max);
```

DESCRIPTION

`PStrNCopy` copies the Pascal string from `src` to `dst`. At most `max` chars are copied. `CStrNCopy` copies up to `max` characters from `src` to `dst` C strings. If `src` string is shorter than `max`, `dst` string will be padded with null characters. If `src` string is longer than `max`, `dst` string will not be null terminated.

StrCat

```
StringPtr PStrCat (StringPtr dst, ConstStr255Param src);

char *CStrCat (char *dst, const char *src);
```

DESCRIPTION

`PStrCat` appends characters from `src` to `dst` Pascal strings. `CStrCat` appends characters from `src` to `dst` C strings. The initial character of `src` overwrites the null character at the end of `dst`. A terminating null character is always appended.

StrNCat

```
StringPtr PStrNCat (StringPtr dst, ConstStr255Param src,UInt32 max);

char *CStrNCat (char *dst, const char *src, UInt32 max);
```

DESCRIPTION

`PStrNCat` appends up to `max` characters from `src` to `dst` Pascal strings. `CStrNCat` appends up to `max` characters from `src` to `dst` C strings. The initial character of `src` overwrites the null character at the end of `dst`. A terminating null character is always appended. Thus, the maximum length of `dst` could be `CStrLen`(`dst`)+`max`+1.

StrCmp

```
short PStrCmp (ConstStr255Param str1, ConstStr255Param str2);

short CStrCmp (const char *str1, const char *str2);
```

DESCRIPTION

`PStrCmp` and `CStrCmp` compare the Pascal and C strings `str1` and `str2` by comparing the values of corresponding characters in each string. These functions treat variations of case, diacritical marks, or other localization factors as different characters.

RETURN CODES

`str1` less than `str2`	–1
`str1` equals `str2`	0
`str1` greater than `str2`	1

StrNCmp

```
short PStrNCmp(ConstStr255Param str1, ConstStr255Param str2, UInt32 max);

short CStrNCmp (const char *str1, const char *str2, UInt32 max);
```

DESCRIPTION

PStrNCmp and CStrNCmp compare the first max C and Pascal strings str1 and str2 by comparing the values of corresponding characters in each string. These functions treat variations of case, diacritical marks, or other localization factors as different characters.

RETURN CODES

str1 less than str2	–1
str1 equals str2	0
str1 greater than str2	1

StrLen

```
UInt32 PStrLen (ConstStr255Param src);

UInt32 CStrLen (const char *src);
```

DESCRIPTION

CStrLen returns the length of the C string src in characters. This does not include the terminating null character. PStrLen returns the length of the Pascal string src in characters.

PStrToCStr and CStrToPStr

```
void PStrToCStr (char *dst, const Str255 src);

void CStrToPStr (Str255 dst, const char *src);
```

DESCRIPTION

PStrToCStr and CStrToPStr convert Pascal strings to C strings and vice versa.

Debugging Support

The following debugging functions are available to driver writers.

```
void SysDebug           (void);
void SysDebugStr        (StringPtr str);
```

DESCRIPTION

`SysDebug` lets you enter the system debugger. `SysDebugStr` lets you enter the system debugger and display the Pascal string pointed to by `str`.

EXECUTION CONTEXT

The debugging routines may be called from task level, software interrupt level, or hardware interrupt level.

Service Limitations

Table 11-2 lists the DSL routines that can be called at the different interrupt levels described in "Device Driver Execution Contexts" (page 346). A dot (•) in the column indicates that the service is available at that level.

The righthand column in Table 11-2 identifies memory allocation services. These services can be called only from task level, and not from a software interrupt. Memory allocation and deallocation can occur when a native driver processes the any of following commands:

`Close`

`Initialize`

`Finalize`

`Open`

`Replace`

`Superseded`

The Name Registry routines `RegistryPropertyGet` and `RegistryPropertyGetSize` are available at secondary interrupt level. All other Name Registry routines are available only at task level.

Applications can freely use the Name Registry and the Driver Loader Library, but with the current release of Mac OS only drivers should use the Driver Services Library.

IMPORTANT
It is the responsibility of the driver writer to conform to these limitations; code that violates them will not work with future releases of Mac OS. ▲

Table 11-2 Services available to drivers

Routine	Task level	Secondary interrupt level	Hardware interrupt level	Memory allocation
`AbsoluteDeltaToDuration`	•	•	•	
`AbsoluteDeltaToNanoseconds`	•	•	•	
`AbsoluteToDuration`	•	•	•	
`AbsoluteToNanoseconds`	•	•	•	
`AddAbsoluteToAbsolute`	•	•	•	
`AddAtomic`	•	•	•	
`AddAtomic8`	•	•	•	
`AddAtomic16`	•	•	•	
`AddDurationToAbsolute`	•	•	•	
`AddNanosecondsToAbsolute`	•	•	•	
`BitAndAtomic`	•	•	•	
`BitAndAtomic8`	•	•	•	
`BitAndAtomic16`	•	•	•	
`BitOrAtomic`	•	•	•	

Table 11-2 Services available to drivers (continued)

Routine	Task level	Secondary interrupt level	Hardware interrupt level	Memory allocation
`BitOrAtomic8`	●	●	●	
`BitOrAtomic16`	●	●	●	
`BitXorAtomic`	●	●	●	
`BitXorAtomic8`	●	●	●	
`BitXorAtomic16`	●	●	●	
`BlockCopy`	●	●	●	
`BlockMove`	●	●	●	
`BlockMoveData`	●	●	●	
`BlockMoveDataUncached`	●	●	●	
`BlockMoveUncached`	●	●	●	
`BlockZero`	●	●	●	
`BlockZeroUncached`	●	●	●	
`CallSecondaryInterruptHandler2`	●	●		
`CancelTimer`	●	●	●	
`ChangeInterruptSetOptions`	●			
`CheckpointIO`	●	●		
`CompareAndSwap`	●	●	●	
`CreateInterruptSet`	●			
`CreateSoftwareInterrupt`	●	●		
`CStrCat`	●	●	●	
`CStrCmp`	●	●	●	
`CStrCopy`	●	●	●	
`CStrLen`	●	●	●	
`CStrNCat`	●	●	●	

Table 11-2 Services available to drivers (continued)

Routine	Task level	Secondary interrupt level	Hardware interrupt level	Memory allocation
CStrNCopy	●	●	●	
CStrToPStr	●	●	●	
CurrentExecutionLevel	●	●	●	
CurrentTaskID	●			
DecrementAtomic	●	●	●	
DecrementAtomic8	●	●	●	
DecrementAtomic16	●	●	●	
DelayFor	●			
DelayForHardware	●	●	●	
DeleteSoftwareInterrupt	●	●		
DeviceProbe	●			
DurationToAbsolute	●	●	●	
DurationToNanoseconds	●	●	●	
FlushProcessorCache	●	●	●	
GetDataCacheLineSize	●	●	●	
GetInterruptFunctions	●			
GetInterruptSetOptions	●			
GetIOCommandInfo	●	●		
GetLogicalPageSize	●	●	●	
GetPageInformation	●			
GetTimeBaseInfo	●	●	●	
IncrementAtomic	●	●	●	
IncrementAtomic8	●	●	●	
IncrementAtomic16	●	●	●	

Table 11-2 Services available to drivers (continued)

Routine	Task level	Secondary interrupt level	Hardware interrupt level	Memory allocation
`InstallInterruptFunctions`	●			
`IOCommandIsComplete`	●	●		
`MemAllocatePhysicallyContiguous`	●			●
`MemDeallocatePhysicallyContiguous`	●			●
`NanosecondsToAbsolute`	●	●	●	
`NanosecondsToDuration`	●	●	●	
`PBDequeue`	●	●	●	
`PBDequeueFirst`	●	●	●	
`PBDequeueLast`	●	●	●	
`PBEnqueue`	●	●	●	
`PBEnqueueLast`	●	●	●	
`PBQueueCreate`	●			
`PBQueueDelete`	●			
`PBQueueInit`	●			
`PoolAllocateResident`	●			●
`PoolDeallocate`	●			●
`PrepareMemoryForIO`*	●			●
`PStrCat`	●	●	●	
`PStrCmp`	●	●	●	
`PStrCmp`	●	●	●	
`PStrCopy`	●	●	●	
`PStrLen`	●	●	●	
`PStrNCat`	●	●	●	
`PStrNCmp`	●	●	●	

Table 11-2 Services available to drivers (continued)

Routine	Task level	Secondary interrupt level	Hardware interrupt level	Memory allocation
`PStrNCopy`	●	●	●	
`PStrToCStr`	●	●	●	
`QueueSecondaryInterruptHandler`	●	●	●	
`RegistryPropertyGet`	●	●		
`RegistryPropertyGetSize`	●	●		
`RegistryPropertySet`†	●			
`SendSoftwareInterrupt`	●	●		
`SetInterruptTimer`	●	●	●	
`SetProcessorCacheMode`	●			
`SubAbsoluteFromAbsolute`	●	●	●	
`SubDurationFromAbsolute`	●	●	●	
`SubNanosecondsFromAbsolute`	●	●	●	
`SynchronizeIO`	●	●	●	
`SysDebug`	●	●	●	
`SysDebugStr`	●	●	●	
`TestAndClear`	●	●	●	
`TestAndSet`	●	●	●	
`UpTime`	●	●	●	

* May be called from a native driver's `DoDriverIO` routine and from any subroutine called from `DoDriverIO`
† The size of the property must not change.

C H A P T E R 1 2

Expansion Bus Manager

This chapter describes a number of services for PCI cards, collectively called the **Expansion Bus Manager,** that are included in the firmware and system software in the second generation of Power Macintosh computers. It is divided into the following major sections:

- "Expansion ROM Contents" summarizes the attributes of a Macintosh compatible PCI card.
- "Nonvolatile RAM" (page 442), illustrates how nonvolatile RAM is allocated in a typical Power Macintosh computer.
- "PCI Nonmemory Space Cycle Generation" (page 453), lists routines that you can use to access memory in the various PCI address spaces.
- "Card Power Controls" (page 469), describes calls that Mac OS uses to control PCI card power levels.

Expansion ROM Contents

The expansion ROM on a PCI card for Macintosh computers must conform to the format and information content defined in Chapter 6 of the PCI specification. The following notes apply to the required device identification fields when used with Macintosh computers:

- The vendor ID must be the identification assigned by the PCI Special Interest Group.
- The device and revision IDs must be assigned by the vendor and need not be registered with Apple.
- The header type and class codes must conform to those specified in the *PCI Local Bus Specification,* Revision 2.0.

Nonvolatile RAM

Power Macintosh computers that support the PCI bus contain at least 8 KB of **nonvolatile RAM (NVRAM)**. The NVRAM can be flash ROM, or RAM powered by the computer's local battery, so that it retains data between system startups. This section describes typical NVRAM configurations and discusses how you can store device properties in NVRAM.

Typical NVRAM Structure

A typical example of allocating 8 KB of NVRAM memory space in a Power Macintosh computer built prior to Macintosh computers that implement the RAM in ROM NewWorld architecture is shown in Table 12-1. The memory space is partitioned into functional areas.

Table 12-1 Typical NVRAM space allocations

Length (bytes)	Description
4096	Operating-system partition
768	Reserved by Apple for diagnostics
256	Reserved by Apple for **parameter RAM**
1024	Reserved by Apple for Name Registry properties
2048	Open Firmware partition

The allocations shown in Table 12-1 provide permanent configuration data storage, both for Mac OS and for PCI expansion cards. The sections that follow describe how this storage is typically used.

Operating-System Partition

The first 4 KB of NVRAM space in a typical configuration may be reserved for use by operating systems other than Mac OS. The Macintosh firmware and system software does nothing with this space except to initialize the first 2 bytes to show that the available NVRAM size is 4 KB.

Note
Operating systems that use this space would need to provide their own protocols for allocating fields and for defining, updating, and checking data. In particular, they would need to follow rules for determining whether fields in the NVRAM operating-system partition use big-endian or little-endian addressing. ◆

Apple-Reserved Partitions

Apple typically reserves 2048 bytes of NVRAM space for use by Macintosh firmware and system software, as shown in Table 12-1. Part of this allocation constitutes the 256 bytes of parameter RAM (PRAM) that all Macintosh computers have traditionally provided for use by Mac OS.

Card firmware and application software can access some of the Macintosh PRAM space by using the Macintosh Toolbox routines described in *Inside Macintosh: Operating System Utilities.*

Open Firmware Partition

The remaining 2048 bytes of NVRAM space is used by the Open Firmware startup process to support PCI expansion cards.

The `little-endian?` variable, discussed in “Addressing Mode Determination” (page 63), is stored in the Open Firmware NVRAM space.

Using NVRAM to Store Name Registry Properties

NVRAM can be used to store device properties permanently. However, such storage is necessary only for devices used during Mac OS startup, because other devices can store an unlimited amount of permanent information on disk in the Preferences folder.

If the `kRegPropertyValueIsSavedToNVRAM` modifier of a property entry is set, the contents of that property entry will be preserved in NVRAM. During Mac OS startup, the Macintosh firmware will retrieve the entry value from NVRAM and place it in the device tree. This modifier is described in “Data Structures and Constants” (page 326).

Properties stored in NVRAM are available to boot devices before the devices have been installed. For example, properties stored in NVRAM can be used to configure a primary display or to define the net address of a network boot device. In both cases, the device driver can access user-changeable information before disk storage services are available.

To provide facilities for multiple boot devices, each node in the Name Registry can store a single, small property in NVRAM. Depending on the version of the Name Registry the format of the information is different. Versions of the Name Registry prior to version 1.2 use the following format to store NVRAM properties:

- device location (6 bytes), an absolute location within the PCI hardware I/O space of the current machine. The format of this value is not public, and its value is not visible to drivers. The device location is not used in Macintosh computers that support version 1.2 of the Name Registry.
- property name (4 bytes), a 1-byte to 4-byte string that is a creator ID assigned by Apple Developer Technical Support. Creator IDs are assigned on a first-come, first-served basis and form unique labels for products such as applications and driver files. You can register a creator ID with Apple Developer Technical Support. In version 1.2 of the Name Registry, the proper name can be a string up to an 8-bytes on length, and does not have to be a creator ID, but the name can incorporate a creator ID.
- property value is a value that is stored by `RegistryPropertySet` or `RegistryPropertyCreate` (provided `kRegPropertyValueIsSavedToNVRAM` is set) and is retrieved by `RegistryPropertyGet`.The name is 8 bytes maximum for versions of the Name Registry prior to 1.2. In version 1.2 of the Name Registry the property value can be up to 32 bytes in size.

There are two ways to determine of the version 1.2 Name Registry rules apply. Try and create a property that doesn't follow the rules, for example create an 8-byte name. If it succeeds, you know the 1.2 rules apply. Or, you can check the version number of the Name Registry.

Note
Using a creator ID (instead of a generic mnemonic) as the name of an NVRAM property value offers protection against acquiring the wrong value when a user configures a system and then moves a hardware device to a different slot or bus. If all drivers define their NVRAM property names with unique creator IDs, a driver can determine whether an NVRAM value is owned by its device. ◆

Use the Name Registry routines described in Chapter 10 to access nodes saved to NVRAM. The Macintosh firmware will return an error message if a driver or application performs one of the following illegal actions:

- Tries to store two properties in NVRAM for the same node. The application should enumerate its properties, fetch the property modifier, and remove incorrect (unknown) properties or clear their NVRAM bits.
- Tries to store more than 8 bytes in an NVRAM property.
- Specifies a property name longer than 4 bytes in versions of the Name Registry prior to 1.2, 8 bytes in version 1.2 or later of the Name Registry.

Because only a single property may be stored in NVRAM for each device, drivers will need to protect themselves against accessing an old NVRAM property that may already be in place. The recommended algorithm is as follows:

1. Iterate to find all properties for the device.
2. If a property has the NVRAM modifier bit set, then check the property name.
3. If the property name is correct, use the property value.
4. If the property name is incorrect, delete the property and use default settings.
5. Exit and use the found property value. Use default settings if no property was set or an incorrectly named property was deleted.

Listing 12-1 shows four sample routines that are useful when manipulating NVRAM:

- `RetrieveDriverNVRAMParameter` retrieves the single property stored in Macintosh NVRAM and checks it.
- `GetDriverNVRAMProperty` looks at a driver property in NVRAM. This routine can be called outside an initialization context.
- `UpdateDriverNVRAMProperty` updates a driver property in NVRAM.
- `CreateDriverNVRAMProperty` creates a driver property that is stored in NVRAM.

Listing 12-1 Sample NVRAM manipulation code

```
#define CopyOSTypeToCString(osTypePtr, resultString) do {      \
       BlockCopy(osTypePtr, resultString, sizeof (OSType));    \
       resultString[sizeof (OSType)] = 0;                      \
   } while (0)

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * RetrieveDriverNVRAMParameter retrieves the single property stored in nonvolatile
 * memory (NVRAM). By convention, this property is named using our registered
 * creator code. Because the PCI system stores properties indexed by physical slot
 * number, we may retrieve an incorrect property if the user moves cards around.
 * To protect against this, we check the property name.
 *
 * This function must be called from an initialization context.
 *
 * Return status:
 * noErr          Success: the NVRAM property was retrieved.
 * nrNotFoundErr  Failure: there was no NVRAM property.
 * paramErr       Failure: there was an NVRAM property, but not ours.
 * other          Failure: unexpected Name Registry error.
 */
OSErr
RetrieveDriverNVRAMProperty(
       RegEntryIDPtr          regEntryIDPtr,     /* driver's Name Registry ID   */
       OSType                 driverCreatorID,   /* registered creator code      */
       UInt8                  driverNVRAMRecord[8]
   )
{
       OSErr                  status;
       RegPropertyIter        cookie;
       RegPropertyNameBuf     propertyName;
       RegPropertyValueSize   regPropertyValueSize;
       RegPropertyModifiers   propertyModifiers;
       Boolean                done;
       char                   creatorNameString[sizeof (OSType) + 1];

       /*
        * search our properties for one with the NVRAM modifier set
        */
       status = RegistryPropertyIterateCreate(regEntryIDPtr, &cookie);
```

```
if (status == noErr) {
    while (status == noErr) {
        /*
         * Get the next property and check its modifier. Break if this is the
         * NVRAM property (there can be only one for our entry ID).
         */
        status = RegistryPropertyIterate(&cookie, propertyName, &done);
        if (status == noErr && done == FALSE) {
            status = RegistryPropertyGetMod(
                        regEntryIDPtr,
                        propertyName,
                        &propertyModifiers
                    );
            if (status == noErr
             && (propertyModifiers & kRegPropertyValueIsSavedToNVRAM) != 0)
                break;
        }
        /*
         * There was no NVRAM property. Return nrNotFoundErr by convention.
         */
        if (status == noErr && done)
            status = nrNotFoundErr;
    }
    RegistryPropertyIterateDispose(&cookie);
    /*
     * If status == noErr, we have found an NVRAM property. Now,
     *  1. If it is our creator code, we have found the property, so
     *      we retrieve the values and store them in the driver's globals.
     *  2. If it was found with a different creator code, the user has
     *      moved our card to a slot that previously had a different card
     *      installed, so delete this property and return (noErr) to use
     *      the factory defaults.
     *  3. If it was not found, the property was not set, so we exit
     *      (noErr); the caller will have preset the values to
     *      "factory defaults."
     */
    if (status == noErr) {
        /*
         * Cases 1 or 2, check the property.
         */
        CopyOSTypeToCString(&driverCreatorID, creatorNameString);
```

```
            if (CStrCmp(creatorNameString, propertyName) == 0) {    /* Match    */
                status = RegistryPropertyGetSize(
                            regEntryIDPtr,
                            propertyName,
                            &regPropertyValueSize
                        );
                if (status == noErr
                 && regPropertyValueSize == sizeof driverNVRAMRecord) {
                    status = RegistryPropertyGet(
                                regEntryIDPtr,
                                propertyName,
                                driverNVRAMRecord,
                                &regPropertyValueSize
                            );
                }
            }
            else {
                /*
                 * This is an NVRAM property, but it isn't ours. Delete the
                 * property and return an error status so the caller uses
                 * "factory settings"
                 */
                status = RegistryPropertyDelete(
                            regEntryIDPtr,
                            propertyName
                        );
                /*
                 * Since we're returning an error anyway, we ignore the
                 * status code.
                 */
                status = paramErr;
            }
        }
    }
    return (status);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Get the NVRAM property. Return an error if it does not exist, is the wrong size,
 * or is not marked "store in NVRAM." This may be called from a
 * noninitialization context.
 * Errors:
 * nrNotFoundErr       Not found in the registry
 * nrDataTruncatedErr  Wrong size
 * paramErr            Not marked "store in NVRAM"
 */
OSErr
GetDriverNVRAMProperty(
       RegEntryIDPtr           regEntryIDPtr,         /* driver's Name Registry ID */
       OSType                  driverCreatorID,       /* registered creator code   */
       UInt8                   driverNVRAMRecord[8]   /* mandated size             */
   )
{
       OSErr                   status;
       char                    creatorNameString[sizeof (OSType) + 1];
       RegPropertyValueSize    size;
       RegPropertyModifiers    modifiers;

       CopyOSTypeToCString(&driverCreatorID, creatorNameString);
       status = RegistryPropertyGetSize(
           regEntryIDPtr,
           creatorNameString,
           &size
       );
       if (status == noErr && size != sizeof driverNVRAMRecord)
           status = nrDataTruncatedErr;
       if (status == noErr) {
           status = RegistryPropertyGetMod(
                       regEntryIDPtr,
                       creatorNameString,
                       &modifiers
                   );
       }
       if (status == noErr
        && (modifiers & kRegPropertyValueIsSavedToNVRAM) == 0)
           status = paramErr;
       if (status == noErr) {
           status = RegistryPropertyGet(
                       regEntryIDPtr,
                       creatorNameString,
                       driverNVRAMRecord,
                       &size
```

```
                );
        }
        return (status);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Update the NVRAM property. Return an error if it was not created. This may be
 * called from PBStatus (or other noninitialization context).
 */
OSErr
UpdateDriverNVRAMProperty(
        RegEntryIDPtr           regEntryIDPtr,       /* driver's Name Registry ID   */
        OSType                  driverCreatorID,      /* registered creator code    */
        UInt8                   driverNVRAMRecord[8]  /* mandated size              */
   )
{
        OSErr                   status;
        char                    creatorNameString[sizeof (OSType) + 1];

        CopyOSTypeToCString(&driverCreatorID, creatorNameString);
        /*
         * Replace the current value of the property with its new value. In this
         * example, we are replacing the entire value and, potentially, changing
         * its size. In production software, you may need to read an existing
         * property and modify its contents. This shouldn't be too hard to do.
         */
        status = RegistryPropertySet(       /* update existing value */
                    regEntryIDPtr,
                    creatorNameString,
                    driverNVRAMRecord,
                    sizeof driverNVRAMRecord
                );
        return (status);
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Create the NVRAM property. This must be called from the driver
 * initialization function.
 */
OSErr
CreateDriverNVRAMProperty(
        RegEntryIDPtr           regEntryIDPtr,        /* driver's Name Registry ID  */
        OSType                  driverCreatorID,      /* registered creator code    */
        UInt8                   driverNVRAMRecord[8]  /* mandated size              */
    )
{
        OSErr                   status;
        char                    creatorNameString[sizeof (OSType) + 1];
        RegPropertyValueSize    size;
        RegPropertyModifiers    modifiers;

        CopyOSTypeToCString(&driverCreatorID, creatorNameString);
        /*
         * Does the property currently exist (with the correct size)?
         */
        status = RegistryPropertyGetSize(       /
* returns noErr if the property exists */
                    regEntryIDPtr,
                    creatorNameString,
                    &size
                );
        if (status == noErr) {
            /*
             * Replace the current value of the property with its new value. In this
             * example, we are replacing the entire value and, potentially, changing
             * its size. In production software, you may need to read an existing
             * property and modify its contents. This shouldn't be too hard to do.
             */
            status = RegistryPropertySet(       /* update existing value */
                        regEntryIDPtr,
                        creatorNameString,
                        driverNVRAMRecord,
                        sizeof driverNVRAMRecord
                    );
        }
        else {
            status = RegistryPropertyCreate(    /* make a new property */
                        regEntryIDPtr,
                        creatorNameString,
                        driverNVRAMRecord,
                        sizeof driverNVRAMRecord
                    );
        }
        /*
         * If status equals noErr, the property has been stored; set its "save to
         * nonvolatile RAM" bit.
         */
        if (status == noErr) {
            status = RegistryPropertyGetMod(
                        regEntryIDPtr,
                        creatorNameString,
                        &modifiers
                    );
        }
        if (status == noErr) {
            /*
             * Set the NVRAM bit and update the modifiers.
             */
            modifiers |= kRegPropertyValueIsSavedToNVRAM;
            status = RegistryPropertySetMod(
                        regEntryIDPtr,
                        creatorNameString,
                        modifiers
                    );
        }
        return (status);
}
```

PCI Nonmemory Space Cycle Generation

"PCI Host Bridge Operation" (page 39), describes how the Macintosh implementation of PCI supports ordinary memory access cycles. Because some PCI cards may need to use other types of PCI cycles—I/O, configuration, interrupt acknowledge, or special cycles—the Expansion Manager includes routines that create these cycle types. These routines are described in the next sections.

All of the nonmemory access routines use the type `RegEntryIDPtr` to identify device nodes in the device tree, as described in Chapter 10, "Name Registry." Drivers should use the `RegEntryIDPtr` value passed to them when they were initialized. Using the `RegEntryIDPtr` type lets the system software determine the target device's location in the device tree, select the appropriate PCI bus to access the device, and generate the correct cycle on that bus.

Fast I/O Space Cycle Generation

The PCI property `assigned-addresses` provides vector entries that represent the physical addresses of devices on expansion cards. Apple has added another property—`AAPL,address`—that provides a vector of 32-bit logical address values, where the *n*th value corresponds to the *n*th `assigned-addresses` vector entry. When accessing device functions located in memory space, you should use the corresponding `AAPL,address` property as the device's base. The same technique is recommended when you are accessing high-performance device functions in I/O space.

Using the `AAPL,address` property, a driver can find the logical address of an I/O resource. Accessing the logical address generates an I/O cycle on the PCI bus. Using the logical base address, a driver can generate a PCI I/O cycle in the same way it accesses a PCI device in memory space. This provides the fastest possible interface to I/O space. For sample code that illustrates this technique, see Listing 9-3 (page 274).

To access a register in memory or I/O space using an `AAPL,address` property, do the following:

1. At initialization, resolve the `assigned-addresses` and `AAPL,address` properties.
2. Search the assigned-addresses vector for an address range in I/O space.
3. Store the corresponding `AAPL,address` vector entry in a variable such as

   ```
   volatile UInt16 *gIORegisterBase;
   ```

4. To read the (16-bit) register at offset 0x04, you can then do

   ```
   value = gIORegisterBase[0x04 / sizeof (UInt16)];
   ```

As with memory accesses, you will need to byte swap the returned value to obtain a Macintosh big-endian result. Byte swapping routines are described on (page 469).

IMPORTANT

Between PCI I/O accesses, software must call the `SynchronizeIO` function (described on (page 373)) to ensure that the accesses affect the PCI device in the correct order. ▲

Slow I/O Space Cycle Generation

Alternatively, you can use the Expansion Bus Manager routines described in this section. They provide byte swapping, enforce in-order execution, and a node-based interface. These extra services add overhead; therefore, for transfer-intensive accesses, such as accessing FIFOs located in I/O space, it is better to use the logical address from the `AAPL,address` property.

The rest of this section describes six routines that let you read and write data to specific I/O addresses, using the base address found in the `assigned-addresses` property (not `AAPL,address`).

ExpMgrIOReadByte

You can use the `ExpMgrIOReadByte` function to read the byte value at a specific address in PCI I/O space.

```
OSErr ExpMgrIOReadByte (
                    RegEntryIDPtr node,
                    LogicalAddress ioAddr,
                    UInt8 *valuePtr);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOReadByte` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`valuePtr`	The returned 8-bit value.

DESCRIPTION

The `ExpMgrIOReadByte` function reads the byte at the I/O address for device node `node` determined by address `ioAddr`, returning its byte-swapped value in `valuePtr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrIOReadWord

You can use the `ExpMgrIOReadWord` function to read the word value at a specific address in PCI I/O space.

```
OSErr ExpMgrIOReadWord (
                    RegEntryIDPtr node,
                    LogicalAddress ioAddr,
                    UInt16 *valuePtr);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOReadWord` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`valuePtr`	The returned 16-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrIOReadWord` function reads the word at the I/O address for device node `node` determined by address `ioAddr`, returning its byte-swapped value in `valuePtr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrIOReadLong

You can use the `ExpMgrIOReadLong` function to read the long word value at a specific address in PCI I/O space.

```
OSErr ExpMgrIOReadLong (
                    RegEntryIDPtr node,
                    LogicalAddress ioAddr,
                    UInt32 *valuePtr);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOReadLong` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`valuePtr`	The returned 32-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrIOReadLong` function reads the long word starting at the I/O address for device node `node` determined by address `ioAddr`, returning its byte-swapped value in `valuePtr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrIOWriteByte

You can use the `ExpMgrIOWriteByte` function to write a byte to an address in PCI I/O space.

```
OSErr ExpMgrIOWriteByte (
                    RegEntryIDPtr node,
                    LogicalAddress ioAddr,
                    UInt8 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOWriteByte` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`value`	The 8-bit value.

DESCRIPTION

The `ExpMgrIOWriteByte` function writes the value of `value` to the I/O address for device node `node` determined by address `ioAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrIOWriteWord

You can use the `ExpMgrIOWriteWord` function to write a word to an address in PCI I/O space.

```
OSErr ExpMgrIOWriteWord (RegEntryIDPtr node,
                         LogicalAddress ioAddr,
                         UInt16 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOWriteWord` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`value`	The 16-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrIOWriteWord` function writes the byte-swapped value of `value` to the I/O address for device node `node` determined by address `ioAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrIOWriteLong

You can use the `ExpMgrIOWriteLong` function to write a long word to an address in PCI I/O space.

```
OSErr ExpMgrIOWriteLong (
                    RegEntryIDPtr node,
                    LogicalAddress ioAddr,
                    UInt32 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrIOWriteLong` returns a result code of `deviceTreeInvalidNodeErr`.
`ioAddr`	The sum of the `assigned-addresses` base address of the device plus the offset to the desired I/O address.
`value`	The 32-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrIOWriteLong` function writes the byte-swapped value of `value` to the I/O address for device node `node` starting at address `ioAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

Configuration Space Cycle Generation

The Expansion Bus Manager contains six routines that let you read and write data to specific addresses in the PCI configuration space for a specified device tree node.

All of the configuration space access routines use the type RegEntryIDPtr to identify device nodes in the device tree, as described in Chapter 10, "Name Registry." Using RegEntryIDPtr lets the system software and the bridge generate the correct PCI configuration cycle for the target device.

ExpMgrConfigReadByte

You can use the ExpMgrConfigReadByte function to read the byte value at a specific address in PCI configuration space.

```
OSErr ExpMgrConfigReadByte (
                    RegEntryIDPtr node,
                    LogicalAddress configAddr,
                    UInt8 *valuePtr);
```

node	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, ExpMgrConfigReadByte returns a result code of deviceTreeInvalidNodeErr.
configAddr	The configuration address (a value between 0 and 255).
valuePtr	The returned 8-bit value.

DESCRIPTION

The ExpMgrConfigReadByte function reads the byte at the address in PCI configuration space for device node node determined by offset configAddr, returning its value in valuePtr.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrConfigReadWord

You can use the `ExpMgrConfigReadWord` function to read the word value at a specific address in PCI configuration space.

```
OSErr ExpMgrConfigReadWord (
                    RegEntryIDPtr node,
                    LogicalAddress configAddr,
                    UInt16 *valuePtr);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrConfigReadWord` returns a result code of `deviceTreeInvalidNodeErr`.
`configAddr`	The configuration address (a value between 0 and 255).
`valuePtr`	The returned 16-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrConfigReadWord` function reads the word at the address in PCI configuration space for device node `node` determined by offset `configAddr`, returning its byte-swapped value in `valuePtr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrConfigReadLong

You can use the `ExpMgrConfigReadLong` function to read the long word value at a specific address in PCI configuration space.

```
OSErr ExpMgrConfigReadLong (
                    RegEntryIDPtr node,
                    LogicalAddress configAddr,
                    UInt32 *valuePtr);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrConfigReadLong` returns a result code of `deviceTreeInvalidNodeErr`.
`configAddr`	The configuration address (a value between 0 and 255).
`valuePtr`	The returned 32-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrConfigReadLong` function reads the long word starting at the address in PCI configuration space for device node `node` determined by offset `configAddr`, returning its byte-swapped value in `valuePtr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrConfigWriteByte

You can use the `ExpMgrConfigWriteByte` function to write a byte to an address in PCI configuration space.

```
OSErr ExpMgrConfigWriteByte (
                          RegEntryIDPtr node,
                          LogicalAddress configAddr,
                          UInt8 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrConfigWriteByte` returns a result code of `deviceTreeInvalidNodeErr`.
`configAddr`	The configuration address (a value between 0 and 255).
`value`	The 8-bit value.

DESCRIPTION

The `ExpMgrConfigWriteByte` function writes the value of `value` to the address in PCI configuration space for device node `node` determined by offset `configAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrConfigWriteWord

You can use the `ExpMgrConfigWriteWord` function to write a word to an address in PCI configuration space.

```
OSErr ExpMgrConfigWriteWord (
                    RegEntryIDPtr node,
                    LogicalAddress configAddr,
                    UInt16 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrConfigWriteWord` returns a result code of `deviceTreeInvalidNodeErr`.
`configAddr`	The configuration address (a value between 0 and 255).
`value`	The 16-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrConfigWriteWord` function writes the byte-swapped value of `value` to the address in PCI configuration space for device node `node` determined by offset `configAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

ExpMgrConfigWriteLong

You can use the `ExpMgrConfigWriteLong` function to write a long word to an address in PCI configuration space.

```
OSErr ExpMgrConfigWriteLong (
                    RegEntryIDPtr node,
                    LogicalAddress configAddr,
                    UInt32 value);
```

`node`	A node identifier that identifies a device node. If you specify a node identifier that isn't in the device tree, `ExpMgrConfigWriteLong` returns a result code of `deviceTreeInvalidNodeErr`.
`configAddr`	The configuration address (a value between 0 and 255).
`value`	The 32-bit value as it would appear on the PCI bus. The function performs the necessary byte swapping.

DESCRIPTION

The `ExpMgrConfigWriteLong` function writes the byte-swapped value of `value` to the address in PCI configuration space for device node `node` starting at offset `configAddr`.

RESULT CODES

`noErr`	0	No error
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

Interrupt Acknowledge Cycle Generation

The routines described in this section generate interrupt acknowledge cycles on the PCI bus. All interrupt acknowledge routines use the type `RegEntryIDPtr` to identify device nodes in the device tree, as described in Chapter 10, “Name Registry.” Using `RegEntryIDPtr` lets the system software and the PCI bridge generate the correct PCI interrupt acknowledge cycle for the target device.

Note
Mac OS does not use PCI interrupt acknowledge cycles. The functionality is provided so that if a PCI device needs an interrupt acknowledge cycle the driver has a way to create the required cycle on the PCI bus. ◆

Interrupt acknowledge cycles for PCI are always read actions. The target node chosen for the functions described in this section should be the single node in the system capable of responding to interrupt acknowledge cycles.

ExpMgrInterruptAcknowledgeReadByte

You can use the `ExpMgrInterruptAcknowledgeReadByte` function to read the byte value resulting from a PCI interrupt acknowledge cycle.

```
OSErr ExpMgrInterruptAcknowledgeReadByte (
                        RegEntryIDPtr entry,
                        UInt8 *valuePtr);
```

`entry`	Pointer to a Name Registry entry ID.
`valuePtr`	Pointer to a buffer to hold the value read.

ExpMgrInterruptAcknowledgeReadWord

You can use the `ExpMgrInterruptAcknowledgeReadWord` function to read the word value resulting from a PCI interrupt acknowledge cycle.

```
OSErr ExpMgrInterruptAcknowledgeReadWord (
                        RegEntryIDPtr entry,
                        UInt16 *valuePtr);
```

`entry`	Pointer to a Name Registry entry ID.

`valuePtr`	Pointer to a buffer to hold the value read.

ExpMgrInterruptAcknowledgeReadLong

You can use the `ExpMgrInterruptAcknowledgeReadLong` function to read the long word value resulting from a PCI interrupt acknowledge cycle.

```
OSErr ExpMgrInterruptAcknowledgeReadLong (
                        RegEntryIDPtr entry,
                        UInt32 *valuePtr);
```

`entry`	Pointer to a Name Registry entry ID.
`valuePtr`	Pointer to a buffer to hold the value read.

Special Cycle Generation

The routines described in this section generate special cycles on the PCI bus.

Some special cycle routines use the type `RegEntryIDPtr` to identify device nodes in the device tree, as described in Chapter 10, “Name Registry.” Using `RegEntryIDPtr` lets the system software and the bridge generate the correct PCI special cycle for the target device.

Note
Special cycles on the PCI bus are broadcast-type cycles. They are always long word write actions. If a node interface is provided, the node chosen for these functions should be behind the bridge that defines the PCI bus on which the special cycle occurs. ◆

ExpMgrSpecialCycleBroadcastLong

You can use the `ExpMgrSpecialCycleBroadcastLong` function to broadcast the long word value in `value` to all PCI buses in the system.

```
OSErr ExpMgrSpecialCycleBroadcastLong (UInt32 value);
```

`value`	The value to be broadcast.

ExpMgrSpecialCycleWriteLong

You can use the `ExpMgrSpecialCycleWriteLong` function to write the long word value in `value` to the PCI bus that contains the device node identified by the name entry pointed to by `entry`.

```
OSErr ExpMgrSpecialCycleWriteLong (
                    RegEntryIDPtr entry,
                    UInt32 value);
```

`entry`	Pointer to a Name Registry entry ID.
`value`	The value to be written.

Byte Swapping Routines

Mac OS provides two routines that help you swap bytes between big-endian and little-endian data formats:

```
UInt16 EndianSwap16Bit (UInt16 data16);

UInt32 EndianSwap32Bit (UInt32 data32);
```

`data16`	2-byte input.
`data32`	4-byte input.

`EndianSwap16Bit` and `EndianSwap32Bit` return byte swapped versions of their input values, thereby converting big-endian data to little-endian or little-endian data to big-endian.

Card Power Controls

If a PCI expansion card normally consumes more than 3 A at 5 V or 2 A at 3.3 V, it should be capable of entering a low-power mode. It is generally useful for all

PCI cards to be able to enter a low-power mode so they will conform to energy-saving standards. Family experts are usually responsible for managing the power consumption characteristics of associated native drivers and may issue power commands or request power information at any time.

A card's driver may elect to ignore power switching commands issued by a family expert by returning the `DriverGestalt` selector `'lpwr'`. Driver Gestalt selectors are defined in Table 8-5 (page 228). It may also return an appropriate indication to the family expert if a switch from high power to low power might interrupt a current or pending operation.

Guidelines

Observe the following power management guidelines for specific classes of drivers:

- As discussed in "Power Services" (page 547), networking drivers should conform to the Open Transport family expert's power management guidelines. The expert handles all interactions with the Power Manager for the driver.
- As discussed in "Graphics Driver Routines" (page 476), graphics drivers should support the `GetSync` and `SetSync` status and control calls to implement the VESA DPMS standard for power management. The Display Manager will handle all interaction with the Power Manager on behalf of the driver.
- SCSI drivers and other classes of drivers for which the family expert interface is not fully defined, or for which a family expert does not currently exist, may need to interact with the Power Manager directly to support power management on PCI-based Power Macintosh computers. However, the current Power Manager interface is not guaranteed to be compatible with future Mac OS releases. Specific issues in this area are discussed in "SCSI Device Power Management" (page 563).

Sample Code

Listing 12-2 shows sample code that retrieves power consumption information from a PCI device.

Listing 12-2 Determining power consumption

```
/*
 * IEEE 1275 defines the "power-consumption" property.
 */
#define kDevicePowerProperty       "power-consumption"
/*
 * Power values are encoded in a vector of "maximum in microwatts." Unspecified
 * valuesshall be zero if other values are provided. Power consumption is 0 for
 * missing values. If the property is missing, the default value will be used.
 */
enum {
   kUnspecifiedStandby,
   kUnspecifiedFullPower,
   kFiveVoltStandby,
   kFiveVoltFullPower,
   kThreeVoltStandby,
   kThreeVoltFullPower,
   kIOPowerStandby,
   kIOPowerFullPower,
   kReservedStandby,
   kReservedFullPower
};

/*
 * The function uses this structure to equate registry entry values with
 * DriverGestalt selectors.
 */
typedef struct PowerInfo {
   OSType              driverGestaltSelector;
   short               correctIndex;
   short               fallbackIndex;
} PowerInfo;
static const PowerInfo gPowerInfo[] = {
   { kDriverGestalt5MaxHighPower,  kFiveVoltFullPower,     kUnspecifiedFullPower   },
   { kDriverGestalt5MaxLowPower,   kFiveVoltStandby,       kUnspecifiedStandby     },
   { kDriverGestalt3MaxHighPower,  kThreeVoltFullPower,    kUnspecifiedFullPower   },
   { kDriverGestalt3MaxLowPower,   kThreeVoltStandby,      kUnspecifiedStandby     },
   { 0,                            0,                      0                       }
};
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Retrieve the driver power consumption vector and search it for the desired power
 * consumption value. Return the desired value, or a default value if the desired
 * value is unavailable. This function does not allocate memory or return any errors.
 */
UInt32
GetDevicePowerConsumption(
       RegEntryIDPtr          regEntryIDPtr,          /
* driver's Name Registry ID   */
       OSType                 driverGestaltSelector,  /
* PBStatus parameter          */
       UInt32                 defaultPowerConsumption /
* default return value        */
   )
{
       OSErr                  status;
       UInt32                 result;
       short                  i;
       short                  index;
       ItemCount              nValues;
       RegPropertyValueSize   size;
       UInt32                 microWatts[kReservedFullPower];

       result = defaultPowerConsumption;
       status = RegistryPropertyGetSize(
                   regEntryIDPtr,
                   kDevicePowerProperty,
                   &size
               );


       if (status == noErr && size <= sizeof microWatts) {
           status = RegistryPropertyGet(
                       regEntryIDPtr,
                       kDevicePowerProperty,
                       (RegPropertyValue *) microWatts,
                       &size
                   );
       }
       if (status == noErr) {
           nValues = size / sizeof microWatts[0];
```

```
            for (i = 0; gPowerInfo[i].driverGestaltSelector != 0; i++) {
                if (gPowerInfo[i].driverGestaltSelector == driverGestaltSelector) {
                    index = gPowerInfo[i].correctIndex;
                    if (index >= nValues)
                        index = gPowerInfo[i].fallbackIndex;
                    if (index < nValues)
                        result = microWatts[index];
                    break;
                }
            }
        }
        return (result);
}
```

C H A P T E R 1 3

Graphics Drivers

This chapter discusses the requirements for designing a native PCI graphics or video display driver for Mac OS on PCI-based Power Macintosh computers. PCI display drivers have a category of `kServiceCategoryNdrvDriver` and a service type of `kNdrvTypeIsVideo`. They export a driver description structure and use the `DoDriverIO` entry point.

For specific information about generic native drivers, see Chapter 8, "Writing Native Drivers." You can also find general information about Macintosh drivers in *Designing Cards and Drivers for the Macintosh Family,* third edition, and *Inside Macintosh: Devices*. These books are listed in "Apple Publications" (page 26). For information about Macintosh pixel formats, see Appendix B, "Graphic Memory Formats."

IEEE Standard 1275 includes graphics extensions that the P1275 Working Group continues to update. For latest information, you can access the FTP site listed in "Institute of Electrical and Electronic Engineers" (page 27).

Apple has revised the way that Macintosh computers automatically sense monitor characteristics. For more information see "Display Timing Modes" (page 506), and *Macintosh New Technical Notes HW-30,* which is available from Apple Developer Support.

Graphics Driver Description

For the Display Manager to load and install a driver, the run-time requirements should be set to `kDriverIsOpenedUponLoad` and `kDriverIsUnderExpertControl`. The device name is used as the name for installation in the unit table. Graphics drivers should report `kServiceCategoryNdrvDriver` as the OS run-time service category and `kNdrvTypeIsVideo` as the type within the category.

A typical driver description structure for a PCI graphics card driver is shown in Listing 8-1 (page 199).

Graphics Driver Routines

In the past, graphics drivers and Mac OS relied on a card's NuBus declaration ROM to get information on the card's capabilities. In PCI-based Power

Macintosh computers, the programming interface for PCI graphics drivers lets the drivers provide the same information to the Mac OS.

Because of potential compilation problems, applications should avoid using high-level Device Manager routines when accessing PCI graphics drivers directly. Use the low-level `PBOpen`, `PBClose`, `PBControlSync`, and `PBStatusSync` routines (described in *Inside Macintosh: Devices*) instead of `FSOpen`, `FSClose`, `Control`, or `Status`.

The next sections detail the specific control and status calls to which a graphics driver must respond.

Control Calls

The following sections present the graphics driver control calls. Not all video or display drivers need to respond to every one of these calls.

Reset (csCode = 0)

The `Reset` routine is obsolete for graphics drivers in the PCI-based Power Macintosh computers. The driver should return `controlErr`.

KillIO (csCode = 1)

The optional `KillIO` routine stops any I/O requests currently being processed and removes any pending I/O requests. If the card does not support asynchronous calls it must return `controlErr`.

SetMode (csCode =2)

The required `SetMode` routine sets the pixel depth of the screen.

```
OSErr = Control(theDeviceRefNum, cscSetMode, &theVDPageInfo );

   -->     csModeDesired   relative bit depth
   --      csData          Unused
   -->     csPage          Desired display page
   <--     csBaseAddr      Base address of video RAM for this csMode
```

To improve the screen appearance during mode changes, devices with settable color tables should set all entries of the Color Lookup table (CLUT) to 50 percent gray before changing the mode. If the video card supports 16-bit or 32-bit pixel depths, the `SetMode` routine should set an internal flag to indicate direct mode operations.

SetEntries (csCode = 3)

The `SetEntries` control routine is required. If the video card is an indexed device, the `SetEntries` control routine should change the contents of the card's CLUT.

```
OSErr = PBControl(theDeviceRefNum, cscSetEntries, &theVDSetEntryRecord);
```

-->	`csTable`	Pointer to ColorSpec array
-->	`csStart`	First entry in table
-->	`csCount`	Number of entries to set

If the value of `csStart` is 0 or positive, the routine must install `csStart` entries starting at that position. If it is –1, the routine must access the contents of the value field in `csTable` to determine which entries are to be changed. Both `csStart` and `csCount` are 0 based—their values are 1 less than the desired amount. For a description of a CLUT and the `ColorSpec` structure, see the Color QuickDraw section of *Inside Macintosh: Imaging With QuickDraw.*

If the card does not have a CLUT (that is, if the `csDeviceType` returned from `GetVideoParameters` does not equal `clutType`), the system should never issue a `SetEntries` control call. If it does, the `SetEntries` control routine should return `controlErr`. With direct devices, the `GrayPage` and `SetGamma` routines are responsible for initializing the hardware properly.

SetGamma (csCode = 4)

The optional `SetGamma` control routine sets the gamma table in the driver that corrects RGB color values.

```
OSErr = Control(theDeviceRefNum, cscSetGamma, &theVDGammaRecord );
```

`--> csGTable` Pointer to gamma table

The gamma table compensates for nonlinearities in a display's color response by providing either a function or a lookup value that associates each displayed color with an absolute RGB value.

To reduce visible flashes resulting from color table changes, the `SetGamma` routine works in conjunction with the `SetEntries` control routine on indexed devices. The `SetGamma` routine first loads new gamma correction data into the driver's private storage; the next `SetEntries` control call applies the gamma correction as it changes the CLUT. `SetGamma` calls are always followed by `SetEntries` control calls on indexed devices.

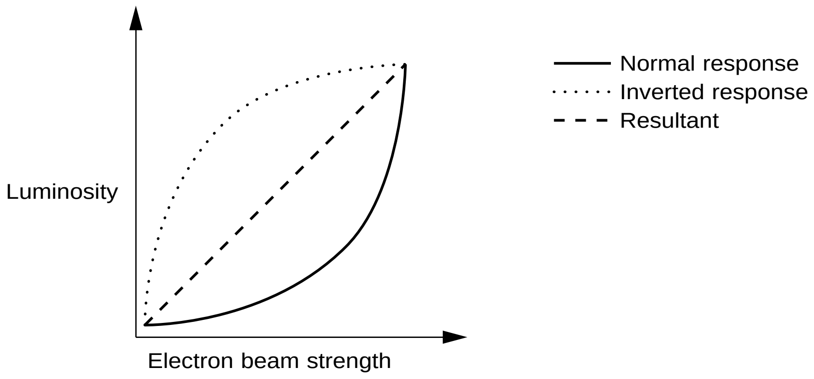
For direct devices, the `SetGamma` routine first sets up the gamma correction data table. Next, it synthesizes a black-to-white linear ramp color table. Finally, it applies the new gamma correction to the color table and sets the data directly in the hardware. Proper correction is particularly important to image-processing applications running on direct devices.

Displays that do not use gamma table correction tend to look oversaturated and dark. Although determining the correct values for a gamma table can be difficult without special tools, the table's contribution to image quality can be striking.

If `NIL` is passed for the `csGTable` value, the driver should build a linear ramp in the gamma table to allow for an uncorrected display.

On a cathode ray tube, phosphors luminesce when they are struck by an electron beam. Unfortunately, there is not a direct correspondence between the luminance of the phosphors and the strength of the electron beam. To create a linear relationship, the actual response is measured and the inverse of its deviation from linearity is applied as a correction factor. Figure 13-1 illustrates this process.

Figure 13-1 Luminosity and electron beam strength

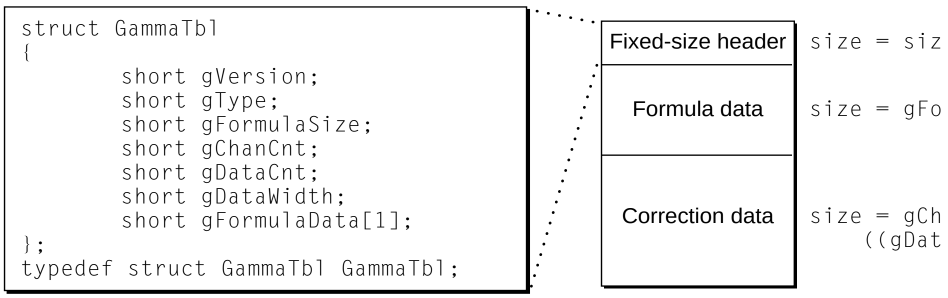


Although this example is described in terms of electron beams and phosphors of a cathode ray tube, similar relationships exist between diode current and LED brightness in active matrix displays.

Gamma Table Implementation

The Power Macintosh gamma table structure is defined in the header file `QuickDraw.h`. Its definition is diagrammed in Figure 13-2.

Figure 13-2 Gamma table structure



```
struct GammaTbl
{
        short gVersion;
        short gType;
        short gFormulaSize;
        short gChanCnt;
        short gDataCnt;
        short gDataWidth;
        short gFormulaData[1];
};
typedef struct GammaTbl GammaTbl;
```

The gamma table is a variable length data structure. As shown in Figure 13-2, the structure `GammaTbl` sits at the front of a pool of memory that holds the data required to apply gamma correction.

The last member of the fixed-length portion of the structure `gFormulaData` is also the entry point to the variable-length portion of the structure. This variable-length portion is divided into two sets, formula data and correction data.

Field descriptions

`gVersion`	The version of the `GammaTbl` data structure. `gVersion == 0` is the only version of the `GammaTbl` data structure currently defined.
`gType`	Since gamma tables are created empirically, they can either attempt to correct the response curve of a specific CLUT, a specific display, or a specific combination of CLUT and display. `gType == 0` indicates that the curve is derived from a display, not a CLUT. In this case, two different hardware modules can share the same gamma table.
`gFormulaSize`	See `gFormulaData`, below.
`gChanCnt`	The number of tables of correction data. If there is more than one channel of correction data, the channels are ordered red, green, blue. If there is only one channel of correction data, the same correction is applied to the red, green, and blue channels of the hardware. The only valid values for `gChanCnt` are 1 and 3.

`gDataCnt`	The number of entries of correction information per channel.
`gDataWidth`	How many significant bits of information are available in each entry, packed to the next larger byte size.
`gFormulaData`	The entry point to the variable-length portion of the gamma table, consisting of the formula data, if any, followed by the correction data. If a gamma table is hardware-invariant (`gType == 0`), then the formula data is never inspected. If a gamma table varies with the hardware (in which case `gType` is the ID of the frame buffer), and `gFormulaSize != 0`, then `gFormulaData[0]` is inspected to see if it is the ID of the monitor currently connected. If the monitor IDs match, the gamma table is considered valid; otherwise it is considered to be the wrong table.

Correction Data

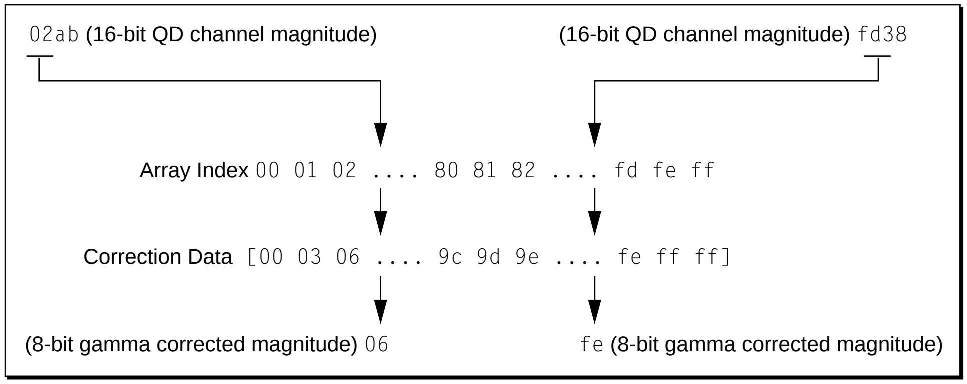
The Correction Data area of the gamma table contains the gamma correction data. If more than one channel's information is present, a block of information for each channel appears in red, green, blue order. There is no field of the `GammaTbl` structure that directly maps to the correction data; instead, correction data is appended to the `gFormulaData` field. To understand how correction data is organized, consider the QuickDraw representation of RGB color:

```
struct RGBColor
{
    unsigned short red;             // magnitude of red channel
    unsigned short green;           // magnitude of green channel
    unsigned short blue;            // magnitude of blue channe
};
typedef struct RGBColor RGBColor;
```

Effectively, the purpose of a gamma table is to map a red, green, or blue channel into another channel. This mapping serves two purposes: to move from 16 bits of significance to `gDataWidth` bits, and to apply luminance correction.

The mapping is usually accomplished by taking the most significant 8 bits of a given channel and using it as an index into that channel's correction data. Two examples of this, with `gDataWidth == 8`, are illustrated in Figure 13-3.

Figure 13-3 Examples of gamma table correction


02ab (16-bit QD channel magnitude)
(16-bit QD channel magnitude) fd38
Array Index 00 01 02 .... 80 81 82 .... fd fe ff
Correction Data [00 03 06 .... 9c 9d 9e .... fe ff ff]
(8-bit gamma corrected magnitude) 06
fe (8-bit gamma corrected magnitude)

Gamma Table Errors

Graphics drivers should return an error code if the following fields of `GammaTbl` do not contain these values:

`gVersion` == 0	This is currently the only defined version of the gamma table structure.
`gType` == 0	This indicates that the gamma table is not dependent on the frame buffer hardware. Few existing gamma tables are frame buffer– specific. This field formerly contained a NuBus construct, `drHWId`, which is no longer applicable.
`gChanCnt` == 1 || `gChanCnt` == 3	Only one or three channels of correction data are supported.

GrayPage (csCode = 5)

The required `GrayPage` routine fills the specified video page with a dithered gray pattern in the current video mode. The page number is 0 based.

```
OSErr = Control(theDeviceRefNum, cscGrayPage, &theVDPageInfo );
```

-- `csMode`	Unused
-- `csData`	Unused
--> `csPage`	Desired display page to gray
-- `csBaseAddr`	Unused

The purpose of the `GrayPage` routine is to eliminate visual artifacts on the screen during mode changes. When the mode changes, the contents of the frame buffer immediately acquire a new color meaning. To avoid annoying color flashes, two events must occur:

- `SetMode` or `SwitchMode` sets the entire contents of the CLUT to 50 percent gray before changing the mode, so that all possible indexes in either the old or new depth appear the same.
- `GrayPage` fills the frame buffer with one of these 50 percent dither patterns:

0xAAAAAAAA represents 32 pixels at 1 bpp

0xCCCCCCCC represents 16 pixels at 2 bpp

0xF0F0F0F0 represents 8 pixels at 4 bpp

0xFF00FF00 represents 4 pixels at 8 bpp

0xFFFF0000 represents 2 pixels at 16 bpp

0xFFFFFFF represents 1 pixel at 32 bpp (invert to get the next pixel)

For direct devices, `GrayPage` also builds a three-channel linear gray color table, gamma-corrects the table, and loads it into the color table hardware.

SetGray (csCode = 6)

The optional `SetGray` routine is used with indexed devices to specify whether subsequent `SetEntries` calls fill a card's CLUT with actual colors or with the luminance-equivalent gray tones.

```
OSErr = Control(theDeviceRefNum, cscSetGray, &theVDGrayRecord );
```

--> `csMode`	Enable or disable luminance mapping

For actual colors (luminance mapping disabled), `SetGray` is passed a `csMode` value of 0; for gray tones (luminance mapping enabled), it is passed a `csMode` value of 1. Luminance equivalence should be determined by converting each RGB value into the hue-saturation-brightness system and then selecting a gray value of equal brightness. Mapping colors to luminance-equivalent gray tones lets a color monitor emulate a monochrome monitor exactly.

If a driver is told to disable luminance mapping and the connected display is known to be a monochrome device, the driver should set `csMode` to 1 and keep luminance mapping enabled.

A direct device should always save the `csMode` value. Luminance mapping, however, should never occur in control routines that modify the CLUT.

SetInterrupt (csCode = 7)

The optional `SetInterrupt` routine controls the generation of VBL interrupts.

```
OSErr = Control(theDeviceRefNum, cscSetInterrupt, &theVDFlag Record );
```

`--> csMode`	Enable or disable interrupts
`-- filler`	Unused

To enable interrupts, pass a `csMode` value of 0; to disable interrupts, pass a `csMode` value of 1. The `VDFlagRecord` data structure is defined on (page 524).

DirectSetEntries (csCode = 8)

`DirectSetEntries` is optional. Normally, color table animation is not used on a direct device, but there are some special circumstances under which an application may want to change the color table hardware. The `DirectSetEntries` routine provides the direct device with indexed mode functionality identical to the regular `SetEntries` control routine.

```
OSErr = PBControl(theDeviceRefNum, cscDirectSetEntries,
                          &theVDSetEntryRecord);
```

`-->`	`csTable`	Pointer to ColorSpec array
`-->`	`csStart`	First entry in table
`-->`	`csCount`	Number of entries to set

The `DirectSetEntries` routine has exactly the same functions and parameters as the regular `SetEntries` routine, but it works only on a direct device. If this call is issued to an indexed device, it should return `controlErr`.

SetDefaultMode (csCode = 9)

The `SetDefaultMode` routine is obsolete for PCI-card graphics drivers in Power Macintosh computers. The driver should return `controlErr`. Graphics drivers should instead use the `SavePreferredConfiguration` routine described on (page 489).

SwitchMode (csCode = 10)

The `SwitchMode` routine is required.

```
OSErr = Control(theDeviceRefNum, cscSwitchMode,
                     &theVDSwitchInfoRecord );
```

`--> csMode`	Relative bit depth to switch to
`--> csData`	DisplayModeID to switch into
`--> csPage`	Video page number to switch into
`<-- csBaseAddr`	Base address of the new DisplayModeID

The `VDSwitchInfoRec` structure, described on (page 523), indicates what depth mode to switch to, the `DisplayModeID` value for the new display mode, and the number of the video page to switch to. The driver uses the `csBaseAddr` field of `VDSwitchInfoRec` to return to the base address of the video page specified by `csPage`.

Note
Unlike NuBus declaration ROM–based drivers, the `SwitchMode` routine should not modify the driver's `AuxDCE` `dCtlSlotId` field. ◆

SetSync (csCode = 11)

The optional `SetSync` routine complements `GetSync`, described on (page 496). It can be used to implement the VESA Device Power Management Standard (DPMS) as well as to enable a sync-on-green, sync-on-red, or sync-on-blue mode for a frame buffer.

```
enum {
    kDisableHorizontalSyncBit = 0,
    kDisableVerticalSyncBit = 1,
    kDisableCompositeSyncBit = 2,
    kEnableSyncOnBlue = 3,
    kEnableSyncOnGreen = 4,
    kEnableSyncOnRed = 5
}
```

The following illustrates a typical use of `SetSync`:

```
OSErr = Control(theDeviceRefNum, cscSetSync, &theVDSyncInfoRec);
```

Following is the information that the status routine must return in the fields of the `VDSyncInfoRec` record (page 496) passed by `SetSync`:

`--> csMode`	Bit map of the sync bits that need to be disabled or enabled.
`--> csFlag`	A mask of the bits that are valid in the `csMode` field. In this manner, a 1 in bit 2 of `csFlag` indicates that bit 2 in the `csMode` field is valid and the driver should set or clear the hardware bit accordingly.

To preserve compatibility with the current Energy Saver control panel, the following special case should be implemented. If the `csFlags` parameter of a `SetSync` routine is 0, the routine should be interpreted as if the `csFlags` parameter were 0x3. This interpretation is necessary because the Energy Saver control panel sends a `csMode` value of 0 and a `csFlags` value of 0 in its parameter block when it wants the display to enable all the horizontal, vertical, and composite sync lines. With the new definition, this would have no effect; the result would be that the display would never come out of sleep mode.

The `SetSync` routine can be used to implement the VESA DPMS standard by disabling the horizontal or vertical sync lines, or both. The VESA DPMS standard specifies four software-controlled modes of operation: On, Standby, Suspend, and Off. Mode switches are accomplished by controlling the horizontal and vertical sync signals. Table 13-1 illustrates the relationship between modes and signals.

Table 13-1 Implementing VESA DPMS modes with `SetSync`

Mode	Horizontal sync	Vertical sync	Video	Power savings	Recovery period
On	Pulses	Pulses	Active	None	n.a.
Standby	No pulses	Pulses	Blanked	Minimal	Short or immediate
Suspend	Pulses	No pulses	Blanked	Significant	Substantial
Off	No pulses	No pulses	Blanked	Maximum	System dependent

In the case of a display using only the composite sync line, only the On and Off modes are possible.

SavePreferredConfiguration (csCode = 16)

The required `SavePreferredConfiguration` routine complements the `GetPreferredConfiguration` control routine described on (page 500). It is used by clients to save the preferred relative bit depth (depth mode) and display mode. This means that a PCI card should save this information in NVRAM so that it persists across system restarts. Note that NVRAM use is limited to 8 bytes. For more information about NVRAM in PCI-based of Power Macintosh computers, see “Typical NVRAM Structure” (page 443).

```
OSErr = Control(theDeviceRefNum, cscSavePreferredConfiguration,
                     &theVDSwitchInfo);
```

The Monitors control panel can use this routine to set the preferred resolution and update the resolution list displayed to the user. Following is the information that the control routine must return in the fields of the `VDSwitchInfoRec` record passed by `SavePreferredConfiguration`:

`--> csMode`	Relative bit depth of preferred resolution
`--> csData`	`DisplayModeID` of preferred resolution

`-- csPage`	Unused
`-- csBaseAddr`	Unused

Note
The driver is not required to save any of the information across reboots. However, it is strongly recommended that the relative bit depth and the `DisplayModeID` value be saved in NVRAM. ◆

SetHardwareCursor (csCode = 22)

`SetHardwareCursor` is a required routine for drivers that support hardware cursors. QuickDraw uses the `SetHardwareCursor` control call to set up the hardware cursor and determine whether the hardware can support it. The driver must determine whether it can support the given cursor and, if so, program the hardware cursor frame buffer (or equivalent), set up the CLUT, and return `noErr`. If the driver cannot support the cursor it must return `controlErr`. The driver must remember whether this call was successful for subsequent `GetHardwareCursorDrawState` or `DrawHardwareCursor` calls, but should not change the cursor's x or y coordinates or its visible state.

```
OSErr = Control (theDeviceRefNum, cscSetHardwareCursor,
                     &theVDSetHardwareCursorRec);
```

`--> csCursorRef`
Reference to cursor data

The driver should call the VSL routine `VSLPrepareCursorForHardwareCursor` with `csCursorRef` and the appropriate hardware cursor descriptor. This routine, described on (page 516), will do all the necessary conversion for the cursor passed in `csCursorRef` to match the hardware described in the hardware cursor descriptor. If the cursor passed in `csCursorRef` is compatible with the hardware cursor descriptor, the VSL call will return `true`; otherwise, it will return `false`. It will also pass back a cursor image at the appropriate bit depth and pixel format for the hardware and a `CTabPtr` color table that specifies the colors for the cursor.

The driver should be able to copy the cursor image passed back from `VSLPrepareCursorForHardwareCursor` directly into its hardware cursor frame buffer (or equivalent) and program its CLUT, using the color table in a fashion similar to the `SetEntries` control call. As in the `SetEntries` control call, the driver must apply any gamma correction to the color table.

If a driver's hardware can support multiple hardware cursor formats, the driver can make multiple calls to `VSLPrepareCursorForHardwareCursor` with different hardware cursor descriptors until the call succeeds or all hardware cursor formats are exhausted.

If the driver must access the cursor data structure passed in `csCursorRef`, it can typecast it to a `CursorImageRec` defined in `Quickdraw.h`. However, the format of the cursor passed in with `csCursorRef` is subject to change in future releases of Mac OS; it is recommended that `VSLPrepareCursorForHardwareCursor` be used because it will be kept up to date with the format of `csCursorRef`.

DrawHardwareCursor (csCode = 23)

`DrawHardwareCursor` is a required routine for drivers that support hardware cursors. It sets the cursor's x and y coordinates and visible state. If the cursor was successfully set by a previous call to `SetHardwareCursor`, the driver must program the hardware with the given x, y, and visible parameters and then return `noErr`. If the cursor was not successfully set by the last `SetHardwareCursor` call, the driver must return `controlErr`.

```
OSErr = Control (theDeviceRefNum, cscDrawHardwareCursor,
                    &theVDDrawHardwareCursorRec);
```

`--> csCursorX`	X coordinate
`--> csCursorY`	Y coordinate
`--> csCursorVisible`	true if the cursor must be visible

The client will have already accounted for the cursor's hot spot, so the `csCursorX` and `csCursorY` values are the x and y coordinates of the upper left corner of the cursor image. Depending on the position of the hot spot, the upper left corner may be above or to the left of the visible screen; thus, `csCursorX` and `csCursorY` are signed values. The driver is responsible for ensuring proper clipping if the cursor lies partially off the screen.

If `csCursorVisible` is `false`, the driver must make the cursor invisible; otherwise, the driver must make the cursor visible.

SetPowerState (csCode = 25)

The optional `SetPowerState` routine lets the display hardware be placed in various power states.

```
OSErr = Control(theDeviceRefNum, cscSetPowerState,
                &theVDPowerStateRec );
```

`--> powerState`
Switch display hardware to this state

`<-- powerFlags`
Describes the status of the new state

The `powerState` constants correlate with the VESA Device Power Management Standards. The system pairs `SetPowerState` and `SetSync` calls. The display hardware should only be placed in a low power state if the graphics controller can also place the connected display in a low power state. In other words, never place the display hardware in a low power state that visibly disrupts video if the connected display would remain active after a corresponding `SetSync` call. The driver is responsible for restoring its state when full power is restored.

Set the `kPowerStateNeedsRefreshBit` bit in `powerFlags` if VRAM decays in the new `powerState` condition. When the driver transitions from a `powerState` condition in which VRAM decays to one in which VRAM is stable, the system will refresh the VRAM.

Status Calls

The following sections present the graphics driver status calls. Not all video or display drivers need to respond to every one of these calls.

GetMode (csCode = 2)

The required `GetMode` routine returns the current relative bit depth, page, and base address.

```
OSErr = Status(theDeviceRefNum, cscGetMode, &theVDPageInfo );
```

`<-- csMode`	Current relative bit depth
` -- csData`	Unused
`<-- csPage`	Current display page
`<-- csBaseAddr`	Base address of video RAM for the current `DisplayModeID` and relative bit depth

GetEntries (csCode = 3)

The required `GetEntries` routine returns the specified number of consecutive CLUT entries, starting with the specified first entry.

```
OSErr = PBStatus(theDeviceRefNum, cscGetEntries, &theVDSetEntryRecord );
```

`<-> csTable`	Pointer to ColorSpec array
`--> csStart`	First entry in table
`--> csCount`	Number of entries to set

If gamma correction is used, the values returned may not be the same as the values originally passed by the `SetEntries` control call. If the value of `csStart` is 0 or positive, the routine must return `csCount` entries starting at that position. If the value of `csStart` is –1, the routine must access the contents of the `Value` fields in `csTable` to determine which entries are to be returned. Both `csStart` and `csCount` are 0 based; their values are 1 less than the desired amount.

Although direct devices do not have logical color tables, the `GetEntries` routine should continue to return the current contents of the CLUT, just as it would for an indexed device.

GetPages (csCode = 4)

The required `GetPages` routine returns the total number of video pages available in the current video card mode, not the current page number. This is a counting number and is not 0 based.

```
OSErr = Status(theDeviceRefNum, cscGetPages, &theVDPageInfo );
```

`-- csMode`	Unused
`-- csData`	Unused
`<-- csPage`	Number of display pages available
`-- csBaseAddr`	Unused

GetBaseAddress (csCode = 5)

The required `GetBaseAddress` routine returns the base address of a specified page in the current mode.

```
OSErr = Status(theDeviceRefNum, cscGetBaseAddr, &theVDPageInfo );
```

`-- csMode`	Unused
`-- csData`	Unused
`--> csPage`	Desired page
`<-- csBaseAddr`	Base address of VRAM for the desired page

The `GetBaseAddress` routine lets video pages be written to even when they are not displayed.

GetGray (csCode = 6)

The required `GetGray` routine describes the behavior of subsequent `SetEntries` control calls to indexed devices.

```
OSErr = Status(theDeviceRefNum, cscGetGray, &theVDGrayRecord );
```

`<-- csMode`	Luminance mapping enabled or disabled

The `csMode` parameter returns 0 if luminance mapping is disabled or 1 if it is enabled.

GetInterrupt (csCode = 7)

The optional `GetInterrupt` status routine returns a value of 0 if VBL interrupts are enabled and a value of 1 if VBL interrupts are disabled.

```
OSErr = Status(theDeviceRefNum, cscGetInterrupt, &theVDFlagRecord );
```

`<-- csMode`	Interrupts enabled or disabled
`-- filler`	Unused

The `VDFlagRecord` data structure is defined on (page 524).

GetGamma (csCode = 8)

The `GetGamma` routine returns a pointer to the current gamma table.

```
OSErr = Status(theDeviceRefNum, cscGetGamma, &theVDGammaRecord );
```

`<-- csGTable`	Pointer to gamma table

The calling application cannot preallocate memory because of the unknown size requirements of the gamma data structure.

GetDefaultMode (csCode = 9)

The `GetDefaultMode` control call is obsolete for PCI card graphics drivers. The driver should return `statusErr`. PCI card graphics drivers for Power Macintosh computers use the `GetPreferredConfiguration` routine described on (page 500).

GetCurrentMode (csCode = 10)

The required `GetCurrentMode` routine uses a `VDSwitchInfoRec` structure.PCI graphics drivers return the current `DisplayModeID` value in the `csData` field.

```
OSErr = Status (theRefNum, cscGetCurMode, &theVDSwitchInfoRec );
```

`<-- csMode`	Current relative bit depth
`<-- csData`	DisplayModeID of current resolution
`<-- csPage`	Current page
`<-- csBaseAddr`	Base address of current page

GetSync (csCode = 11)

The use of the optional `GetSync` and `SetSync` routines has been expanded to manage the settings of all synchronization-related parameters of a frame buffer controller, not just the horizontal and vertical syncs. `GetSync` and `SetSync` can be used to implement the VESA DPMS as well as enable a sync-on-green mode for the frame buffer.

A `VDSyncInfoRec` data structure has been defined for the `GetSync` and `SetSync` routines:

```
struct VDSyncInfoRec {
    unsigned char       csMode;
    unsigned char       csFlags;
}
```

The `csMode` parameter specifies the state of the sync lines according to these bit definitions:

```
enum {
    kDisableHorizontalSyncBit = 0,
    kDisableVerticalSyncBit = 1,
    kDisableCompositeSyncBit = 2,
    kEnableSyncOnBlue = 3,
    kEnableSyncOnGreen = 4,
    kEnableSyncOnRed = 5
};
```

To implement the DPMS standard, bits 0 and 1 of the `csMode` field should have the following values:

Bit 1	Bit 0	Status
0	0	Active
0	1	Standby
1	0	Idle
1	1	Off

`GetSync` can be used in two ways: to get the current status of the hardware and to get the capabilities of the frame buffer controller. These two different kinds of information are discussed in the next sections.

Reporting the Frame Buffer Controller's Capabilities

To find out what the frame buffer controller can do with its sync lines, the user of the `GetSync` routine passes a value of 0xFF in the `csMode` flag. The driver zeroes out those bits that represent a feature that is not supported by the frame buffer controller. The available bit values are those for the `csMode` parameter of `VDSyncInfoRec` (page 497).

For example, a driver that is capable of controlling the horizontal, vertical, and composite syncs, and can enable sync on red, would return a value of 0x27:

```
    csMode = 0x0 |
                ( 1 << kDisableHorizontalSyncBit) |
                ( 1 << kDisableVerticalSyncBit) |
                ( 1 << kDisableCompositeSyncBit) |
                ( 1 << kEnableSyncOnRed)
```

An additional bit is defined to represent those frame buffers that are not capable of controlling the individual syncs separately but can control them as a group:

```
enum {
    kNoSeparateSyncControlBit = 6
}
```

A driver that cannot control the syncs separately sets this bit to tell the client that the horizontal, vertical, and composite syncs are not independently controllable and can only be controlled as a group. Using the previous example, the driver reports a `csMode` of 0x47:

```
csMode = 0x0 |
                ( 1 << kDisableHorizontalSyncBit) |
                ( 1 << kDisableVerticalSyncBit) |
                ( 1 << kDisableCompositeSyncBit) |
                ( 1 << kEnableSyncOnRed) |
                ( 1 << kNoSeparateSyncControlBit)
```

Reporting the Current Sync Status

The other use of the `GetSync` status routine is to get the current status of the sync lines. The client passes 0x00 in the `csMode` field. The returned value represents the current status of the sync lines. Bit 6 (`kNoSeparateSyncControlBit`) has no meaning in this case.

GetConnection (csCode = 12)

The required `GetConnection` routine gathers information about the attached display.

```
OSErr = Status (yourDeviceRefNum, cscGetConnection,
                     &theVDDisplayConnectInfoRec);
```

`<-- csDisplayType`
  Display type of attached display

`<-- csConnectTaggedType`
  Type of tagging

`<-- csConnectTaggedData`
  Tagging data

`<-- csConnectFlags`
  Connection flags

`<-- csDisplayComponent`
  Return display component, if available

See “Responding to GetConnectionInfo” (page 507) for more information on how to implement the `GetConnection` routine.

GetModeTiming (csCode = 13)

The `GetModeTiming` routine is required to report timing information for the desired `displayModeID`.

```
OSErr = Status(yourDeviceRefNum, cscGetModeTiming, &theVDTimingInfoRec);
```

`--> csTimingMode`
  Desired `DisplayModeID`

`<-- csTimingFormat`
  Format for timing info (kDeclROMtables)

`<-- csTimingData`
  Scan timing for desired DisplayModeID

`<-- csTimingFlags`
  Report whether this scan timing is optional or required

See “Display Timing Modes” (page 506) for more details on the `GetModeTiming` routine.

GetModeBaseAddress

The `GetModeBaseAddress` call is obsolete in PCI-based Power Macintosh computers. The driver should return `statusErr`.

GetPreferredConfiguration (csCode = 16)

The required `GetPreferredConfiguration` routine complements `SavePreferredConfiguration`, described on (page 489). `GetPreferredConfiguration` returns the data that was set using `SavePreferredConfiguration`.

```
OSErr = Status(theDeviceRefNum, cscGetPreferredConfiguration,
                     &theVDSwitchInfo);
```

`<-- csMode`	Relative bit depth of preferred resolution
`<-- csData`	`DisplayModeID` of preferred resolution
`-- csPage`	Unused
`-- csBaseAddr`	Unused

GetNextResolution (csCode = 17)

The required `GetNextResolution` routine reports all display resolutions that the driver supports.

```
OSErr = Status(theDeviceRefNum, cscGetNextResolution,
                     &theVDResolutionInfoRec);
```

`--> csPreviousDisplayModeID`	ID of the previous display mode
`<-- csDisplayModeID`	ID of the display mode following csPreviousDisplayModeID.

<-- csHorizontalPixels
Number of pixels in a horizontal line

<-- csVerticalLines
Number of lines in a screen

<-- csRefreshRate
Vertical refresh rate of the screen

<-- csMaxDepthMode
Max relative bit depth for this DisplayModeID

GetNextResolution passes a csPreviousDisplayModeID value and returns the next supported display mode. The csDisplayModeID field is updated and the csHorizontalPixels, csVerticalLines, and csRefreshRate fields are set. The csMaxDepthMode field is also set with the highest supported video bit depth. This uses the same convention as in the past; kDepthMode1 is the first relative bit depth supported, not necessarily 1 bit per pixel. For futher information about depth modes, see the next section.

Observe these cautions:

- The DisplayModeID values used do not need to be the same as the ones Apple uses. However, the DisplayModeID value 0 and all values with the high bit set (0x80000000 through 0xFFFFFFFF) are reserved by Apple.
- To get the first resolution supported by a display, the caller will pass a value of kDisplayModeIDFindFirstResolution in the csPreviousDisplayModeID field of the VDResolutionInfoRec structure.
- To get the second resolution, the caller will pass the csDisplayModeID value of the first resolution in the structure’s csPreviousDisplayModeID field.
- When a call has the last supported resolution in the csPreviousDisplayModeID field, the driver should return a value of kDisplayModeIDNoMoreResolutions in the csDisplayModeID field. No error should be returned.
- If an invalid value is passed in the csPreviousDisplayModeID field, the driver should return a paramErr value without modifying the structure.
- If the csPreviousDisplayModeID field is kDisplayModeIDCurrent, the driver should return information about the current displayModeID.

The constants just described are defined in the file Video.h and are listed in “Data Structures” (page 521).

GetVideoParameters (csCode = 18)

The required `GetVideoParameters` routine returns video parameter information.

```
OSErr = Status (theDeviceRefNum, cscGetVideoParameters,
                     &theVDVideoParametersRec);
```

`--> csDisplayModeID`
ID of the desired `DisplayModeID`

`--> csDepthMode`
Relative bit depth

`<-> *csVPBlockPtr`
Pointer to a `VPBlock`

`<-- csPageCount`
Number of pages supported for resolution and relative bit depth

`<-- csDeviceType`
Direct, fixed, or CLUT

The `GetVideoParameters` routine accepts `csDisplayModeID`, `csDepthMode`, and a pointer to a `VPBlock` structure, which it fills in with the data for the specified `csDisplayModeID` and `csDepthMode`. It also returns the `pageCount` for that particular bit depth, as well as the `deviceType`.

Note
In PCI-based graphics drivers, the `csVPBlockPtr->vpBaseOffset` is always 0. The base address of video RAM for the current page, is the `BaseAddress` value returned by the `GetCurrentMode` routine. ◆

GetGammaInfoList (csCode = 20)

The `GetGammaInfoList` routine is optional. Clients wishing to find a graphics card's available gamma tables formerly accessed the Slot Manager data structures. PCI graphics drivers must return this information directly.

In the future, gamma tables will be part of the display's domain, not the graphics driver's domain. In the meantime, graphics drivers must still provide support for them by responding to the `GetGammaInfoList` and `RetrieveGammaTable` calls. The `GetGammaInfoList` routine iterates over the gamma tables supported by the driver for the attached display.

```
OSErr = Status(theDeviceRefNum, cscGetGammaInfoList, &theVDGammaListRec);
```

`--> csPreviousGammaTableID`	ID of the previous gamma table
`<-- csGammaTableID`	ID of the gamma table following `csPreviousDisplayModeID`
`<-- csGammaTableSize`	Size of the gamma table in bytes
`<-- csGammaTableName`	Gamma table name (C string)

The `csGammaTableName` parameter is a C string with a maximum of 31 characters. The driver needs to copy the name from its storage to the storage passed in by the caller. It can use `CStrCopy`, described on (page 430). The caller uses `csGammaTableSize` to allocate storage to read the entire structure, using the `RetrieveGammaTable` routine.

Observe these cautions:

- A client will pass a `csPreviousGammaTableID` of `kGammaTableIDFindFirst` to get the first gamma table ID. The driver should return this value in the `csGammaTableID` field.
- If the last gamma table ID is passed in the `csPreviousGammaTableID` field, the driver should put a `kGammaTableIDNoMoreTables` in the `csGammaTableID` field and return `noErr`.
- If an invalid gamma table ID is passed in the `csPreviousGammaTableID` field, the driver should return `paramErr` and should not modify the data structure.
- A client can pass `csPreviousGammaTableID` with a value of `kGammaTableIDSpecific`. This tells the driver that the `csGammaTableID` contains the ID of the table that the client wants information about. This is a way to bypass iteration through all the tables when the caller already knows the `GammaTableID`.
- Although the `GetGammaInfoList` call appears to perform its iteration operations similarly to the `GetNextResolution` call, there is an important difference. `GetGammaInfoList` only returns information for gamma tables that

are applicable to the attached display; `GetNextResolution` returns the information regardless of what display is connected.

RetrieveGammaTable (csCode = 21)

The optional `RetrieveGammaTable` routine copies the designated gamma table into the designated location.

```
OSErr = Status (theDeviceRefNum, cscRetrieveGammaTable,
                     &theVDRetrieveGammaRec);
```

```
--> csGammaTableID
```
ID of gamma table to retrieve

```
<-> csGammaTablePtr
```
Location to copy table into

`RetrieveGammaTable` is used after a client has used the `GetGammaInfoList` routine to iterate over the available gamma tables and subsequently decides to retrieve one. It is the responsibility of the client to allocate and dispose of the memory pointed to by `csGammaTablePtr`.

SupportsHardwareCursor (csCode = 22)

Graphics drivers that support hardware cursors must return `true` in response to the `SupportsHardwareCursor` status call.

```
OSErr = Status (theDeviceRefNum, cscSupportsHardwareCursor,
                     &theVDSupportsHardwareCursorRec);
```

```
<-- csSupportsHardwareCursor
```
`true` if hardware cursor is supported

GetHardwareCursorDrawState (csCode = 23)

`GetHardwareCursorDrawState` is a required routine for drivers that support hardware cursors.

```
OSErr = Status (theDeviceRefNum, cscGetHardwareCursorDrawState,
                     &theVDHardwareCursorDrawStateRec);
```

`<-- csCursorX`
: X coordinate from last `DrawHardwareCursor` call

`<-- csCursorY`
: Y coordinate from last `DrawHardwareCursor` call

`<-- csCursorVisible`
: true if the cursor is visible

`<-- csCursorSet`
: true if cursor was successfully set by the last `SetHardwareCursor` call

The `csCursorSet` parameter should be `true` if the last `SetHardwareCursor` control call was successful and `false` otherwise. If `csCursorSet` is true, the `csCursorX`, `csCursorY`, and `csCursorVisible` values must match the parameters passed in to the last `DrawHardwareCursor` control call.

After driver initialization the cursor's visible state and set state should be `false`. After a mode change the cursor should be made invisible but the set state should remain unchanged.

GetPowerState (csCode = 25)

The optional `GetPowerState` routine reports the display hardware's current power state.

```
OSErr = Status (theDeviceRefNum, cscGetPowerState, &theVDPowerStateRec );
```

`<-- powerState`
: Current power state of display hardware

`<-- powerFlags`

Status of current state

Set `kPowerStateNeedsRefreshBit` in `powerFlags` if VRAM decays in the current power state.

Display Timing Modes

Macintosh graphics drivers have always sensed the type of display attached to the graphics card. They did this with three lines on the connector to perform a hardware sense code algorithm. This algorithm is detailed in the *Macintosh New Technical Note HW-30,* described in "Apple Publications" (page 26). Once the sense code was determined, the graphics driver trimmed its list of available timing modes to those that it calculated were possible.

Having the driver determine which timing modes are possible is very unflexible. New displays have required new sense codes that old drivers do not recognize and new technologies, such as the Display Data Channel (DDC) technology, provide additional information that old drivers do not know how to interpret.

Thus, the graphics driver strategy for Mac OS changed in PCI-based Power Macintosh computers. This new strategy emphasizes timing mode decisions done through the Display Manager instead of the graphics driver. This approach has these advantages:

- It gives display designers maximum flexibility to create displays that support multiple timing modes.
- It lets card desgners focus on hardware and be less concerned with the display that is attached.
- It supports the Video Electronics Standards Association (VESA) DDC standard (Level 2B), but does not force cards to interpret DDC content.

Display Manager Requirements

The Display Manager needs support from the graphics driver in order to implement the trimming of the available timing modes. In the past, the driver has trimmed these modes depending on the display that was sensed. Now the driver must perform the following functions:

- Report as available (that is, do not trim) all timing modes that are supported by the current graphics card hardware—for example, trim only those modes that require different amounts or configurations of VRAM. When responding to `GetNextResolution` calls, the driver must return all timing modes supported by the current frame buffer. Do this for DDC displays, multiple scan displays, and single-mode displays.
- If an unknown sense code is found, program the hardware as if a 13- or 14-inch Monitor were sensed.
- If no display is sensed, return an error code from the `Initialize` or `Open` routine.
- When responding to `GetModeTiming`, report as not valid and not safe those timing modes not validated by the sensing algorithm. Do this by clearing the `modeValid` and `modeSafe` flags.
- When responding to `GetConnectionInfo`, perform the extended sense algorithm specified in the next section.
- Support for DDC.

Note
The reason for reporting invalid modes is that the Display Manager interfaces with smart displays and allows those displays to adjust the valid and safe flags monitor by monitor. The card has to know less about the actual capabilities of the display, and the display manufacturer has more flexibility about which modes will be active. ◆

Responding to GetConnectionInfo

The `GetConnectionInfo` call has been modified to support the new monitor sensing scheme described in the previous section. Specifically, changes have been made to a previously reserved field. This section describes the new functionality that graphics drivers need to support to be compatible with the new timing mode trimming procedure.

New Field and Bit Definitions

The `csConnectTagged` field, an unsigned short, in the previous definition has been split into two fields, `csConnectTaggedType` and `csConnectTaggedData`:

```
struct VDDisplayConnectInfoRec {
    unsigned short  csDisplayType;          /* type of display */
    unsigned char   csConnectTaggedType;    /* type of tagging */
    unsigned char   csConnectTaggedData;    /* tagging data */

    unsigned long   csConnectFlags;         /* tells about the */
                                            /* connection */
    unsigned long   csDisplayComponent;     /* if the card has a */
                        /* direct connection to the display, it */
                        /* returns the display component here (future) */
    unsigned long   csConnectReserved;  /* reserved*/
};
```

These two new fields are used to report monitor sensing information, as long as the bit `kTaggingInfoNonStandard` of the `csConnectFlags` field is not set (see next section). If that bit is set, then the `csConnectTaggedType` and `csConnectTaggedData` fields are private and Mac OS will not interpret them. Following are the bit definitions for the `csConnectFlags` field:

```
    enum (
        kAllModesValid              = 0,
        kAllModesSafe               = 1,
        kReportsTagging             = 2,    // driver reports tagging
        kHasDirectConnection        = 3,
        kIsMonoDev                  = 4,
        kUncertainConnection        = 5,
        kTaggingInfoNonStandard     = 6,
        kReportsDDCConnection       = 7,
        kHasDDCConnection           = 8
    };
```

Reporting csConnectTaggedType and csConnectTaggedData

`GetConnectionInfo` is designed to be a real-time call, particularly when it is used for tagging. When a driver receives this call, it should read the sense lines, obtaining the raw sense code and the extended sense code.

IMPORTANT

The driver is required to do this every time it gets this call. It cannot just report the codes it sensed during initialization. ▲

When the `kTaggingInfoNonStandard` bit of `csConnectFlags` is cleared to 0, then `csConnectTaggedType` and `csConnectTagged` data are used to report the raw sense code and the extended sense code, respectively.

The following enumeration shows the constants used for `csConnectTaggedType` when `kTaggingInfoNonStandard` is 0:

```
typedef unsigned char RawSenseCode;
enum {
    kRSCZero        = 0,
    kRSCOne         = 1,
    kRSCTwo         = 2,
    kRSCThree       = 3,
    kRSCFour        = 4,
    kRSCFive        = 5,
    kRSCSix         = 6,
    kRSCSeven       = 7
};
```

The `RawSenseCode` data type contains constants for the possible raw sense code values when “standard” sense code hardware is implemented. For such sense code hardware, the raw sense is obtained as follows:

- Instruct the frame buffer controller not to drive any of the monitor sense lines actively.
- Read the state of the monitor sense lines 2, 1, and 0. Line 2 is the MSB, 0 the LSB.

IMPORTANT

When the `kTaggingInfoNonStandard` bit of `csConnectFlags` is `false`, then the `RawSenseCode` constants are valid `csConnectTaggedType` values in `VDDisplayConnectInfo`. ▲

The following enumeration shows the constants used for `csConnectTaggedData` when `kTaggingInfoNonStandard` is 0:

```
typedef unsigned char ExtendedSenseCode;
enum {
    kESCZero21Inch              = 0x00,     /* 21" RGB */
    kESCOnePortraitMono         = 0x14,     /* portrait monochrome */
    kESCTwo12Inch               = 0x21,     /* 12" RGB */
    kESCThree21InchRadius       = 0x31,     /* 21" RGB (Radius) */
    kESCThree21InchMonoRadius   = 0x34,     /* 21" monochrome (Radius) */
    kESCThree21InchMono         = 0x35,     /* 21" monochrome */
    kESCFourNTSC                = 0x0A,     /* NTSC */
    kESCFivePortrait            = 0x1E,     /* Portrait RGB */
    kESCSixMSB1                 = 0x03,     /* Multiscan band-1 (12" */
                                            /* thru 16") */
    kESCSixMSB2                 = 0x0B,     /* Multiscan band-2 (13" */
                                            /* thru 19") */
    kESCSixMSB3                 = 0x23,     /* Multiscan band-3 (13" */
                                            /* thru 21") */
    kESCSixStandard             = 0x2B,     /* 13" or 14" RGB or 12" */
                                            /* monochrome*/
    kESCSevenPAL                = 0x00,     /* PAL */
    kESCSevenNTSC               = 0x14,     /* NTSC */
    kESCSevenVGA                = 0x17,     /* VGA */
    kESCSeven16Inch             = 0x2D,     /* 16" RGB (GoldFish) */
    kESCSevenPALAlternate       = 0x30,     /* PAL (alternate) */
    kESCSeven19Inch             = 0x3A,     /* Third-party 19” */
    kESCSevenNoDisplay          = 0x3F      /* No display connected */
};
```

The `ExtendedSenseCode` data type contains enumerated constants for the values that are possible when the extended sense algorithm is applied to hardware that implements the “standard” sense code algorithm.

For such sense code hardware, the algorithm is as follows, where sense line A corresponds to 2, B to 1, and C to 0:

- Drive sense line A low and read the values of B and C.
- Drive sense line B low and read the values of A and C.
- Drive sense line C low and read the values of A and B.

In this way, a 6-bit number of the form BC/AC/AB is generated.

IMPORTANT

When the `kTaggingInfoNonStandard` bit of `csConnectFlags` is `false`, then these constants are valid `csConnectTaggedData` values in `VDDisplayConnectInfo`. ▲

Table 13-2 shows examples of `csConnectTaggedType` and `csConnectTaggedData` values for certain monitors.

Table 13-2 Sample `csConnectTaggedType` and `csConnectTaggedData` values

Display	`csConnectTaggedType`	`csConnectTaggedData`
21" Apple RGB	0	0x00
20" Apple Multiscan	6	0x23
14" Apple RGB	6	0x2B

Connection Information Flags

The following values have been added to the connection information flags to supply required information to the Display Manager:

- `kReportsDDCConnection` = 7 means that the card supports the DDC and would report a connection if a DDC display were connected.
- `kHasDDCConnection` = 8 means the card has a DDC connection to the display.
- `kTaggingInfoNonStandard` = 5 means that the information reported in `csConnectTaggedType` and `csConnectTaggedData` fields does not correspond to the Apple sense codes.

The flag `kHasDirectConnect` has been renamed `kHasDirectConnection`.

Timing Information

The file `Video.h` contains constants for Apple-defined timings. A driver returns the timing for a given display mode by `GetTimingInfo`. The `csTimingData` field of the `VDTimingInfoRec` contains the timing constant for the display mode. The Display Manager and smart monitors use it to adjust the valid and safe flags. The `VDTimingInfoRec` structure is described on (page 523).

Timing information should reflect the actual timing driving the display. For example, even if a card creates a large graphics device with hardware pan and zoom for a 13-inch RGB display, it should still return `timingApple13`.

Some Apple displays support display modes such as 640x480 on a 16-inch display. The display is being driven at 16-inch timing, but the graphics device is built smaller. The timing information for that display mode should still be `timingApple16`.

Reporting Display Resolution Values

In the NuBus environment, the driver's primary initialization routine trims the supported display resolutions (functional sResources) to those that are available on the display that is sensed. This makes it difficult to support new displays, as possible supported resolutions might have been deleted by the card's primary initialization routine. The Display Manager now takes care of verifying that a particular resolution is supported by the current display, using `GetModeConnection` and `GetTimingInfo`.

The following sections detail what the different routines should do to implement the reporting of all possible display resolutions. See the previous section, "Display Timing Modes" (page 506), for background information on timing modes.

Implementing the GetNextResolution Call

A driver should leave all modes (resolutions) supported by the current video card hardware (for example, trim the modes that correspond to different amounts of VRAM). The driver should do this for all displays, even single-mode displays. This will help to decouple the graphics driver from knowing the capabilities of new displays.

Implementing the GetModeConnection Call

The Display Manager uses `GetModeConnection` to ascertain the capabilities of a connected display. For this call, the driver should not attempt to determine whether the various modes are valid or safe. This means the `kAllModesValid` and `kAllModesSafe` bits of the `csConnectFlags` field should be set to 0. By setting these bit fields to 0, the driver forces the Display Manger to make a `GetModeTiming` status call for each timing mode instead of assuming that they all have the same state.

Implementing the GetModeTiming Call

`GetModeTiming` is used by the Display Manager to gather scan timing information. If the driver does not believe the display is capable of being driven with the desired resolution, it marks the `kModeValid` and `kModeSafe` bits of the `csTimingFlags` field `false`. This indicates to the Display Manager that the driver doesn't think the display can handle the resolution but will let the Display Manager make the final decision, possibly by asking another software module for more information.

Programming the Hardware

A graphics driver should program the hardware to a valid and safe resolution, according to the sensed display. It should still report data as defined in the previous sections. The driver could also program the hardware to its previous resolution (before the last system restart), assuming that this information is valid for the current display.

Supporting the Hardware Cursor

PCI-based Power Macintosh computers implement a hardware cursor capability that graphics drivers may support. The status and control calls that graphics drivers implementing hardware cursors must respond to are as following:

- `SupportsHardwareCursor` status call (`csCode` = 22), described on (page 504)
- `GetHardwareCursorDrawState` status call (`csCode` = 23), described on (page 505)
- `SetHardwareCursor` control call (`csCode` = 22), described on (page 490)
- `DrawHardwareCursor control call` (`csCode` = 23), described on (page 491)

Only drivers that provide a hardware cursor need to respond to these calls.

A utility routine, `VSLPrepareCursorForHardwareCursor`, helps drivers convert QuickDraw's internal cursor representation into their hardware cursor's format. This routine is described in "Hardware Cursor Utility" (page 516).

Video Services Library

The Macintosh Video Services Library (VSL) provides video interrupt services for vertical blanking, horizontal blanking, and other tasks. It also contains a utility that can be used by graphics drivers that respond to hardware cursor calls as described in "Supporting the Hardware Cursor" (page 513).

Interrupt Services

This section describes functions in the VSL that help video drivers signal the Macintosh software to service display interrupts associated with the display attached to the frame buffer.

A driver can create as many interrupt services as it supports. The model described here supports different types of video interrupts, such as horizontal blanking and frame interrupts. It opens the door for specialized interrupts for specific applications (such as broadcast). For each queue it supports, the driver is responsible for calling `VSLDoInterruptService` when the associated interrupt happens.

VSLNewInterruptService

```
OSErr VSLNewInterruptService(RegEntryIDPtr serviceOwner,
                     InterruptServiceType serviceType,
                     InterruptServiceId* serviceID);
```

`serviceOwner`
`RegEntryIDPtr` passed to the driver at install time.

`serviceType` Type of interrupt to be created.

`serviceID` Returned to specify the service for further calls to the VSL.

```
typedef unsigned long               InterruptServiceId;
typedef ResType                     InterruptServiceType;
enum {
    kVBLService = 'vbl ';           // vertical blanking
    kHBLService = 'hbl ';           // horizontal blanking
    kFrameService = 'fram';         // interlace mode
};
```

DESCRIPTION

`VSLNewInterruptService` creates a new interrupt for a graphics device. The service owner is the `RegEntryIDPtr` value passed to the driver at install time. This is used to identify the owner. The service type is a `resType` value indicating the type of interrupt to be created. At this time only one interrupt of a given type can be created by a driver. The `serviceID` value is returned by VSL and is used to specify the service for any further calls to VSL.

`VSLNewInterruptService` can be called only at driver install, open, and close times—times when memory management calls are safe.

VSLDoInterruptService

```
OSErr VSLDoInterruptService( InterruptServiceId serviceID );
```

`serviceID`	Value returned by `VSLNewInterruptService`.

DESCRIPTION

`VSLDoInterruptService` executes tasks associated with an interrupt service. When a graphics driver gets an interrupt, it determines which service corresponds to that interrupt and calls `VSLDoInterruptService` with the `serviceID` value for that service. `VSLDoInterruptService` executes any tasks associated with the service.

VSLDisposeInterruptService

```
OSErr VSLDisposeInterruptService( InterruptServiceId serviceID );
```

`serviceID` Value returned by `VSLNewInterruptService`.

DESCRIPTION

`VSLDisposeInterruptService` disposes of an interrupt service. When a graphics driver is closing for good, so that the card interrupt will no longer be serviced, it should call `VSLDisposeInterruptService`. The VSL will take over servicing any tasks still in the service.

`VSLDisposeInterruptService` can only be called at driver install, open, and close times—times when memory management calls are safe.

Hardware Cursor Utility

Drivers that support hardware cursors are passed a reference to a cursor stored in QuickDraw's internal representation. This cursor format must be converted into the hardware cursor's format. This conversion could include translating bit depths, interpreting the cursor mask, and matching colors.

To facilitate support for hardware cursors, the VSL provides a utility routine that performs the cursor conversion. By setting up a record that describes the hardware cursor's format, a driver can call this routine to do the conversion for it.

VSLPrepareCursorForHardwareCursor

```
Boolean VSLPrepareCursorForHardwareCursor
                    (void *cursorRef,
                    HardwareCursorDescriptorPtr hardwareDescriptor,
                    HardwareCursorInfoPtr hwCursorInfo);
```

`cursorRef` Reference to the cursor passed in by QuickDraw.

`hardwareDescriptor`
Hardware cursor format.

`hwCursorInfo` Passed back to the driver to program the hardware cursor.

DESCRIPTION

If the `cursorRef` passed to the driver is capable of being rendered by the hardware cursor, `VSLPrepareCursorForHardwareCursor` returns `true`; otherwise, it returns `false`. Cases where the routine returns `false` include a cursor needing more colors than the hardware can supply, a cursor that is too big, and a cursor requiring special pixel types that the hardware doesn't support, such as inverted pixels.

The driver uses the following structure to describe its hardware cursor:

```
enum {
    kTransparentEncoding            = 0,
    kInvertingEncoding
};

enum {
    kTransparentEncodingShift           = (kTransparentEncoding << 1),
    kTransparentEncodedPixel         = (0x01 <<
kTransparentEncodingShift),
    kInvertingEncodingShift             = (kInvertingEncoding << 1),
    kInvertingEncodedPixel              = (0x01 <<
kInvertingEncodingShift),
};

enum {
    kHardwareCursorDescriptorMajorVersion = 0x0001,
    kHardwareCursorDescriptorMinorVersion = 0x0000
};

struct HardwareCursorDescriptorRec {
    UInt16                      majorVersion;
    UInt16                      minorVersion;
    UInt32                      height;
    UInt32                      width;
    UInt32                      bitDepth;
    UInt32                      maskBitDepth;
    UInt32                      numColors;
    UInt32                      *colorEncodings;
    UInt32                      flags;
    UInt32                      supportedSpecialEncodings;
    UInt32                      specialEncodings[16];
};

typedef struct HardwareCursorDescriptorRec HardwareCursorDescriptorRec,
*HardwareCursorDescriptorPtr;
```

The `majorVersion` and `minorVersion` fields describe the version of the descriptor record. The driver must set these to `kHardwareCursorDescriptorMajorVersion` and `kHardwareCursorDescriptorMinorVersion`. Doing so will provide compatibility with the conversion routine if the descriptor is changed in future releases of the VSL.

The `height` and `width` fields specify the maximum cursor height and width, in pixels, supported by the hardware.

The `bitDepth` field specifies the bit depth of the hardware cursor.

The `maskBitDepth` field is currently unused but reserved for future use. The driver must set this field to 0.

The `numColors` field specifies the number of colors supported by the hardware.

The `colorEncodings` field points to an array that specifies the hardware pixel encodings that map to the colors in the hardware cursor color table. The first entry in this array specifies the hardware cursor pixel value that corresponds to the first entry in the hardware cursor's color table; the second entry in this array specifies the pixel value for the second entry in the hardware's color table, and so on.

The `flags` field is used for extra information about the hardware. Currently, all flag bits are reserved and must be set to 0.

The `supportedSpecialEncodings` field specifies the type of special pixels supported by the hardware cursor and how they're implemented.

The special pixel types supported by the descriptor are transparent pixels and inverting pixels. Transparent pixels are invisible, and the frame buffer pixel underneath a transparent hardware cursor pixel is seen. Inverting hardware cursor pixels invert the frame buffer pixel underneath.

The `specialEncodings` field is an array that specifies the pixel values for special encodings. Use the constants `kTransparentEncoding` and `kInvertingEncoding` to index into the array.

EXAMPLES

The following hardware descriptor specifies a typical two-color hardware cursor:

```
UInt32  cursorColorEncodings[] =
{
    0, 1
};

HardwareCursorDescriptorRec  hardwareCursorDescriptor =
{
    kHardwareCursorDescriptorMajorVersion,  // major version number
    kHardwareCursorDescriptorMinorVersion,  // minor version number
    32,                                     // height
    32,                                     // width
    2,                                      // pixel depth
    0,                                      // mask depth
    2,                                      // number of cursor colors
    &cursorColorEncodings,                  // color pixel encodings
    0,                                      // flags
    kTransparentEncodedPixel |          // supports transparent pixels
    kInvertingEncodedPixel,                 // supports inverting pixels
    2,                                      // transparent pixel encoding
    3,                                      // inverting pixel encoding
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 // unused encodings
}
```

The foregoing describes a 2-bit-per-pixel hardware cursor that can be up to 32 by 32 pixels in size and supports transparent and inverting pixels. A cursor pixel value of 0 will display the first color in the cursor's color map, and a pixel value of 1 will display the second color. A cursor pixel value of 2 will display the color of the screen pixel underneath the cursor. A cursor pixel value of 3 will display the inverse of the color of the screen pixel underneath the cursor.

The following hardware descriptor describes a three-color hardware cursor:

```
UInt32 cursorColorEncodings[] =
{
    1, 2, 3
};
```

```
HardwareCursorDescriptorRec  hardwareCursorDescriptor =
{
    kHardwareCursorDescriptorMajorVersion,  // major version number
    kHardwareCursorDescriptorMinorVersion,  // minor version number
    32,                                    // height
    32,                                    // width
    2,                                     // pixel depth
    0,                                     // mask depth
    3,                                     // number of cursor colors
    &cursorColorEncodings,                 // color pixel encodings
    0,                                     // flags
    kTransparentEncodedPixel,              // supports transparent pixels
    0,                                     // transparent pixel encoding
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 // unused encodings
};
```

The foregoing describes a 2-bit-per-pixel hardware cursor that can be up to 32 by 32 pixels in size and supports transparent pixels. A cursor pixel value of 1 displays the first color in the cursor's color map, a pixel value of 2 displays the second color, and a pixel value of 3 displays the third color. A cursor pixel value of 0 displays the color of the screen pixel underneath the cursor. If the cursor requires inverting pixels (for example, the I-beam text edit cursor), a call to `VSLPrepareCursorForHardwareCursor` will return `false` and the driver should let the cursor be implemented in software.

The `VSLPrepareCursorForHardwareCursor` call will return the information that the driver needs to program the hardware cursor in the following data structure:

```
enum {
    kHardwareCursorInfoMajorVersion            = 0x0001,
    kHardwareCursorInfoMinorVersion            = 0x0000
};

struct HardwareCursorInfoRec {
    UInt16                      majorVersion;
    UInt16                      minorVersion;
    UInt32                      cursorHeight;
    UInt32                      cursorWidth;
    CTabPtr                     colorMap;
    Ptr                         hardwareCursor;
    UInt32                      reserved[6];
};

typedef struct HardwareCursorInfoRec HardwareCursorInfoRec,
                        *HardwareCursorInfoPtr;
```

The `majorVersion` and `minorVersion` fields describe what version of the info record is being used. The driver must set these to `kHardwareCursorInfoMajorVersion` and `kHardwareCursorInfoMinorVersion`. Doing so will provide compatibility with the conversion routine if the descriptor is changed in future releases of the VSL.

The `cursorHeight` and `cursorWidth` fields specify the height and width of the cursor passed in from QuickDraw.

The `colorMap` field is the table of colors that the cursor uses. A table big enough to hold all of the colors supported by the hardware cursor must be passed to the `VSLPrepareCursorForHardwareCursor` call, which will fill this table with the appropriate colors. These colors are taken from the color table in the `gDevice` record for the driver's display. The driver must perform any required gamma correction on this color table.

The `hardwareCursor` field points to the buffer containing the converted image for the hardware cursor. A buffer big enough to hold the largest cursor supported by the hardware must be passed to the `VSLPrepareCursorForHardwareCursor` call, which will fill this buffer with the appropriate pixel values. The conversion call will not necessarily fill the entire buffer if the cursor passed from QuickDraw is smaller than the largest cursor supported by the hardware. The `hardwareCursor` buffer image's row bytes will equal `cursorWidth` times the pixel depth of the hardware cursor. The driver must set the extra pixels to be transparent.

The `reserved` field is an array of reserved values, and the driver must set these to 0.

Data Structures

Mac OS uses the data structures listed in this section to communicate with graphics drivers. The interface file `Video.h` contains the latest information about these structures.

```
struct VPBlock {
    long    vpBaseOffset;   /*always 0 for Slot Mgr independent drivers*/
    short   vpRowBytes;     /*width of each row of video memory*/
    Rect    pBounds;        /*BoundsRect for the video display */
    short   vpVersion;      /*PixelMap version number*/
    short   vpPackType;
    long    vpPackSize;
    long    vpHRes;         /*horiz res of the device (pixels per inch)*/
    long    vpVRes;         /*vert res of the device (pixels per inch)*/
    short   vpPixelType;    /*defines the pixel type*/
    short   vpPixelSize;    /*number of bits in pixel*/
    short   vpCmpCount;     /*number of components in pixel*/
    short   vpCmpSize;      /*number of bits per component*/
    long    vpPlaneBytes;   /*offset from one plane to the next*/
};
```

In PCI-based graphics drivers, the `vpBaseOffset` is always 0. The base address of video RAM for the current page, is the `BaseAddress` value returned by the `GetCurrentMode` routine.

```
struct VDEntryRecord {
    Ptr     csTable;        /*pointer to color table entry*/
};

struct VDGrayRecord {
    Boolean     csMode;     /*same as GDDevType value (0=color, 1=mono)*/
    SInt8       filler;
};

struct VDSetEntryRecord {
    ColorSpec   *csTable;   /*pointer to an array of color specs*/
    short       csStart;    /*which spec in array to start with, or -1*/
    short       csCount;    /*number of color spec entries to set*/
};

struct VDGammaRecord {
    Ptr         csGTable;   /*pointer to gamma table*/
};
```

```
struct VDSwitchInfoRec {
    UInt16      csMode;         /*relative bit depth*/
    UInt32      csData;         /*display mode ID*/
    UInt16      csPage;         /*page to switch in*/
    Ptr         csBaseAddr;     /*base address of page (return value)*/
    UInt32      csReserved;     /*reserved (set to 0) */
};

struct VDTimingInfoRec {
    UInt32      csTimingMode;       /* timing mode (a la InitGDevice) */
    UInt32      csTimingReserved;   /* reserved */
    UInt32      csTimingFormat;     /* what format is the timing info */
    UInt32      csTimingData;       /* data supplied by driver */
    UInt32      csTimingFlags;      /* information*/
};

struct VDDisplayConnectInfoRec {
    UInt16      csDisplayType;          /* type of display connected */
    UInt8       csConnectTaggedType;    /* type of tagging */
    UInt8       csConnectTaggedData;    /* tagging data */
    UInt32      csConnectFlags;         /* info about the connection */
    UInt32      csDisplayComponent;     /* display component if card has direct */
                                        /* connection to display (future) */
    UInt32      csConnectReserved;      /* reserved */
};

struct VDPageInfo {
    short       csMode;
    long        csData;
    short       csPage;
    Ptr         csBaseAddr;
};

struct VDResolutionInfoRec {
    DisplayModeID       csPreviousDisplayModeID;    /* ID of the previous resolution */
                                                    /* in a chain */
    DisplayModeID       csDisplayModeID;            /* ID of the next resolution */
    unsigned long       csHorizontalPixels;         /* # of pixels in a horizontal */
                                                    /* line at the max depth */
    unsigned long       csVerticalLines;            /* # of lines in a screen at the */
                                                    /* max depth */
    Fixed               csRefreshRate;              /* vertical refresh rate, Hz */
    DepthMode           csMaxDepthMode;             /* 0x80-based max bit depth */
    unsigned long       csResolutionFlags;          /* flag bits */
    unsigned long       csReserved;                 /* reserved */
};

typedef struct VDResolutionInfoRec VDResolutionInfoRec;

/* csResolutionFlags bit flags for VDResolutionInfoRec*/
enum {
    kResolutionHasMultipleDepthSizes = 0
        /* this mode has different csHorizontalPixels, csVerticalLines at */
        /* different depths (usually slightly larger at lower depths) */
};

struct VDVideoParametersInfoRec {
   DisplayModeID   csDisplayModeID;    /* ID of the target resolution */
   DepthMode       csDepthMode;        /* resolution’s relative bit depth */
   VPBlockPtr      csVPBlockPtr;       /* pointer to video parameter block */
   UInt32          csPageCount;        /* number of pages supported by the
                                          resolution */
   VideoDeviceType csDeviceType;       /* direct, fixed, or CLUT */
   UInt32          csReserved;         /* reserved */
};

struct VDFlagRecord {
   SInt8    csMode;                    /* interrupts enabled or disabled */
   SInt8    filler;                    /* reserved */
};

struct VDGetGammaListRec {
   GammaTableID    csPreviousGammaTableID;   /* ID of previous gamma table */
   GammaTableID    csGammaTableID;           /* ID of gamma table following
                                                csPreviousDisplayModeID */
   UInt32      csGammaTableSize;         /* size of gamma table in bytes */
   char        csGammaTableName[32];     /* gamma table name (C string) */
};
```

```
struct VDRetrieveGammaRec {
   GammaTableID   csGammaTableID;     /* ID of gamma table to retrieve */
   GammaTbl       *csGammaTablePtr;   /* location to copy desired gamma to */
};

struct VDSupportsHardwareCursorRec {
   Boolean     csSupportsHardwareCursor;   /* true if HW cursor supported */
   SInt8       filler;
};

struct VDSetHardwareCursorRec {
   void        *csCursorRef;
};

struct VDDrawHardwareCursorRec {
   SInt32      csCursorX;
   SInt32      csCursorY;
   SInt32      csCursorVisible;
};

struct VDSyncInfoRec {
   UInt8       csMode;
   UInt8       csFlags;
};

struct VDConvolutionInfoRec {
   DisplayModeID  csDisplayModeID;    /* ID of resolution we  want info on */
   DepthMode      csDepthMode;        /* Relative bit depth  */
   UInt32         csPage;
   UInt32         csFlags;
   UInt32         csReserved;
};

struct VDPowerStateRec {
    unsigned long       powerState;
    unsigned long       powerFlags;
    unsigned long       powerReserved1;
    unsigned long       powerReserved2;
};
```

```
typedef UInt32    DisplayModeID;
typedef UInt32    VideoDeviceType;
typedef UInt32    GammaTableID;

/*  Power Mode constants for VDPowerStateRec.powerState.*/
    kAVPowerOff,
    kAVPowerStandby,
    kAVPowerSuspend,
    kAVPowerOn
};

enum {
/*  Power Mode constants for VDPowerStateRec.powerFlags.*/
    kPowerStateNeedsRefreshBit= 0,
    kPowerStateNeedsRefreshMask= (1L << 0)
};

/* bit definitions for the get/set sync call*/
enum {
   kDisableHorizontalSyncBit   = 0,
   kDisableVerticalSyncBit     = 1,
   kDisableCompositeSyncBit    = 2,
   kEnableSyncOnBlue           = 3,
   kEnableSyncOnGreen          = 4,
   kEnableSyncOnRed            = 5,
   kNoSeparateSyncControlBit   = 6,
   kHorizontalSyncMask         = 0x01,
   kVerticalSyncMask           = 0x02,
   kCompositeSyncMask          = 0x04,
   kDPMSSyncMask               = 0x7,
   kSyncOnBlueMask             = 0x08,
   kSyncOnGreenMask            = 0x10,
   kSyncOnRedMask              = 0x20,
   kSyncOnMask                 = 0x38
};

/* Bit definitions for the get/set convolution call*/
enum {
   kConvolved            = 0,
   kLiveVideoPassThru     = 1,
   kConvolvedMask          = 0x01,
   kLiveVideoPassThruMask  = 0x02
};

/* csTimingFormat values in VDTimingInfo */
/* timing info follows DeclROM format */
enum {
   kDeclROMtables      = 'decl'
};
enum {
    timingInvalid              = 0,   /* unknown timing; user must confirm*/
    timingApple_512x384_60hz  = 130, /*  512x384  (60 Hz) Rubik timing*/
    timingApple_560x384_60hz  = 135, /*  560x384  (60 Hz) Rubik-560 timing*/
    timingApple_640x480_67hz  = 140, /*  640x480  (67 Hz) HR timing*/
    timingApple_640x400_67hz  = 145, /*  640x400  (67 Hz) HR-400 timing*/
    timingVESA_640x480_60hz   = 150, /*  640x480  (60 Hz) VGA timing*/
    timingApple_640x870_75hz  = 160, /*  640x870  (75 Hz) FPD timing*/
    timingApple_640x818_75hz  = 165, /*  640x818  (75 Hz) FPD-818 timing*/
    timingApple_832x624_75hz  = 170, /*  832x624  (75 Hz) GoldFish timing*/
    timingVESA_800x600_56hz   = 180, /*  800x600  (56 Hz) SVGA timing*/
    timingVESA_800x600_60hz   = 182, /*  800x600  (60 Hz) SVGA timing*/
    timingVESA_800x600_72hz   = 184, /*  800x600  (72 Hz) SVGA timing*/
    timingVESA_800x600_75hz   = 186, /*  800x600  (75 Hz) SVGA timing*/
    timingVESA_1024x768_60hz  = 190, /* 1024x768  (60 Hz) VESA 1K-60Hz*/
    timingVESA_1024x768_70hz  = 200, /* 1024x768  (70 Hz) VESA 1K-70Hz*/
    timingApple_1024x768_75hz = 210, /* 1024x768  (75 Hz) Apple 19" RGB*/
    timingApple_1152x870_75hz = 220, /* 1152x870  (75 Hz) Apple 21" RGB*/
    timingAppleNTSC_ST        = 230, /*  512x384  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingAppleNTSC_FF        = 232, /*  640x480  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingAppleNTSC_STconv    = 234, /*  512x384  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingAppleNTSC_FFconv    = 236, /*  640x480  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingApplePAL_ST         = 238, /*  640x480  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingApplePAL_FF         = 240, /*  768x576  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingApplePAL_STconv     = 242, /*  640x480  (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingApplePAL_FFconv     = 244, /*  768x576 (60 Hz, interlaced,
                                                   nonconvolved)*/
    timingVESA_1280x960_75hz   = 250, /* 1280x960  (75 Hz)*/
    timingVESA_1280x1024_60hz  = 260, /* 1280x1024 (60 Hz)*/
    timingVESA_1280x1024_75hz  = 262, /* 1280x1024 (75 Hz)*/
    timingVESA_1600x1200_60hz  = 280, /* 1600x1200 (60 Hz) VESA proposed*/
    timingVESA_1600x1200_65hz  = 282, /* 1600x1200 (65 Hz) VESA proposed*/
    timingVESA_1600x1200_70hz  = 284, /* 1600x1200 (70 Hz) VESA proposed*/
    timingVESA_1600x1200_75hz  = 286, /* 1600x1200 (75 Hz) VESA proposed*/
    timingVESA_1600x1200_80hz  = 288  /* 1600x1200 (80 Hz) VESA proposed
                                                   (pixel clock is 216 Mhz dot
clock)*/

/* csConnectFlags values in VDDisplayConnectInfo */
enum {
   kAllModesValid          = 0,
   kAllModesSafe           = 1,
   kReportsTagging         = 2,
   kHasDirectConnection    = 3,
   kIsMonoDev              = 4,
   kUncertainConnection    = 5,
   kTaggingInfoNonStandard = 6,
   kReportsDDCConnection   = 7,
   kHasDDCConnection       = 8
};

/* csDisplayType values in VDDisplayConnectInfo */
enum {
    kUnknownConnect             = 1,
    kPanelConnect               = 2,        /* for use with fixed-in-place LCD panels
*/
    kPanelTFTConnect            = 2,        /* alias for kPanelConnect */
    kFixedModeCRTConnect        = 3,        /* for use with fixed-mode
                                               (i.e. very limited range) displays */
    kMultiModeCRT1Connect       = 4,        /* 320x200 maybe, 12" maybe, 13" (default),
                                               16" certain, 19" maybe, 21" maybe */
    kMultiModeCRT2Connect       = 5,        /* 320x200 maybe, 12" maybe, 13" certain,
                                               16"
(default), 19" certain, 21" maybe */
    kMultiModeCRT3Connect       = 6,        /* 320x200 maybe, 12" maybe, 13" certain,
                                               16"
```

```
certain, 19" default, 21" certain */
    kMultiModeCRT4Connect       = 7,        /* expansion to large multimode
                                                (not yet used) */
    kModelessConnect            = 8,        /* expansion to modeless model
                                                (not yet used) */
    kFullPageConnect            = 9,        /* 640x818 (to get 8bpp in 512K case)
                                                and 640x870 (these two only) */
    kVGAConnect                 = 10,       /* 640x480 VGA default--
                                                question everything else */
    kNTSCConnect                = 11,       /* NTSC ST (default), FF, STconv, FFconv */
    kPALConnect                 = 12,       /* PAL ST (default), FF, STconv, FFconv */
    kHRConnect                  = 13,       /* 640x400 (to get 8bpp in 256K case)
                                                and 640x480 (these two only) */
    kPanelFSTNConnect           = 14        /* for use with fixed-in-place LCD FSTN
                                                (aka "Supertwist") panels */
};

/* csTimingFlags values in VDTimingInfoRec */
enum {
    kModeValid              = 0,    /* says that this mode should NOT be trimmed */
    kModeSafe               = 1,    /* this mode does not need confirmation */
    kModeDefault            = 2,    /* default mode for this type connection */
    kModeShowNow            = 3,    /* this mode should always be shown (even
                                        though it may require a confirm) */
    kModeNotResize          = 4,    /* should not be used to resize the display,
                                    e.g. mode selects different connector on card */
    kModeRequiresPan        = 5     /* has more pixels than are actually displayed */
};

typedef unsigned short DepthMode;
enum {
   kDepthMode1 = 128,
   kDepthMode2 = 129,
   kDepthMode3 = 130,
   kDepthMode4 = 131,
   kDepthMode5 = 132,
   kDepthMode6 = 133

typedef unsigned char RawSenseCode;
```

```
enum {
   kRSCZero            = 0,
   kRSCOne             = 1,
   kRSCTwo             = 2,
   kRSCThree           = 3,
   kRSCFour            = 4,
   kRSCFive            = 5,
   kRSCSix             = 6,
   kRSCSeven           = 7
};

typedef unsigned char ExtendedSenseCode;
enum {
    kESCZero21Inch                  = 0x00,        /* 21" RGB          */
    kESCOnePortraitMono             = 0x14,        /* portrait Monochrome      */
    kESCTwo12Inch                   = 0x21,        /* 12" RGB          */
    kESCThree21InchRadius           = 0x31,        /* 21" RGB (Radius)     */
    kESCThree21InchMonoRadius= 0x34,               /* 21" monochrome (Radius)  */
    kESCThree21InchMono             = 0x35,        /* 21" monochrome           */
    kESCFourNTSC                    = 0x0A,        /* NTSC                 */
    kESCFivePortrait                = 0x1E,        /* Portrait RGB         */
    kESCSixMSB1                     = 0x03,        /
* Multiscan band-1 (13" thru 16") */
    kESCSixMSB2                     = 0x0B,        /
* Multiscan band-2 (13" thru 19") */
    kESCSixMSB3                     = 0x23,        /
* Multiscan band-3 (13" thru 21") */
    kESCSixStandard                 = 0x2B,        /* 13"/
14" RGB or 12" Monochrome   */
    kESCSevenPAL                    = 0x00,        /* PAL              */
    kESCSevenNTSC                   = 0x14,        /* NTSC                 */
    kESCSevenVGA                    = 0x17,        /* VGA              */
    kESCSeven16Inch                 = 0x2D,        /* 16" RGB (GoldFish)       */
    kESCSevenPALAlternate           = 0x30,        /* PAL (alternate)      */
    kESCSeven19Inch                 = 0x3A,        /* Third-party 19”      */
    kESCSevenNoDisplay              = 0x3F         /* No display connected         */
};

enum {
    kDisplayModeIDCurrent = 0x0,                   // reference the current
DisplayModeID
```

```
    kDisplayModeIDInvalid = 0xffffffff,             // a bogus DisplayModeID in all
cases
    kDisplayModeIDFindFirstResolution = 0xfffffffe,                 // used in
                                                                    //
GetNextResolution to
                                                                    // reset iterator
    kDisplayModeIDNoMoreResolutions = 0xfffffffd                    // used in
                                                                    //
GetNextResolution to
                                                                    // indicate end of
list
}

enum {
    kGammaTableIDFindFirst = 0xfffffffe,                // get the first gamma table ID
    kGammaTableIDNoMoreTables = 0xfffffffd,             // used to indicate end of list
    kGammaTableIDSpecific = 0x0    // return the info for the given table ID
}
```

Replacing Graphics Drivers

Mac OS is able to replace the ROM-based PCI graphics driver. You can use this feature to fix a bug or add additional functionality that was not found in the ROM-based driver. This section details several guidelines for replacing the driver. Prerequisite information is contained in “Driver Replacement” (page 174).

Note
Replacing a graphics driver may disrupt the user’s experience if the screen flashes or is redrawn. The following discussion suggests ways to prevent or control this. ◆

Starting with version 1.1 of the System Enabler, Mac OS issues a `kSupersededCommand` to the outgoing driver and a `kReplaceCommand` to the new driver. Note that a driver that gets the `kSupersededCommand` will not get a `kFinalizeCommand`. Similarly, the driver getting the `kReplaceCommand` will not get the `kInitializeCommand`.

To implement these new calls, the ROM-based driver must support the `kSuperseded` command. In the call's implementation, the driver must place in the Name Registry any information that will be needed by the new driver. It should not reset the video hardware (for example, by turning the video sync signals off).

A `kCloseCommand` will always be issued before `kFinalizeCommand` or `kSupersedeCommand`. When this command is received, the driver should turn off all interrupts and remove all VSL services. When responding to `kFinalizeCommand` and `kSupersededCommand`, it should remove the interrupt services.

The new driver needs to support the `kReplaceCommand`. After reading the state information from the Name Registry (which the old driver put there), it must make sure that all the current information is correctly initialized in the hardware. When responding to the `kReplaceCommand` it should not reprogram the hardware, because this might make the display flash.

The `kReplaceCommand` routine can ask Mac OS to redraw the screen by creating a property named `needFullInit` in the device node of the Name Registry. On finding that property, the Mac OS will redraw the screen and then delete the property. Redrawing the screen might be required if the new driver needed to change a parameter in the hardware (such as `rowBytes`) that is reflected in the OS data structures.

CHAPTER 14

Network Drivers

This chapter describes what must be done to create STREAMS drivers for the Apple Open Transport networking architecture. It also describes the minimal functionality that must be supported by any driver that works with the Open Transport implementations of AppleTalk and TCP/IP. In this chapter, STREAMS drivers are also called **port drivers.**

Open Transport uses the STREAMS model for implementing protocols and drivers to provide flexibility for mixing and matching protocols. This approach also allows a wide range of third-party STREAMS modules and drivers to be easily ported to the Open Transport environment.

Part of the flexibility of the STREAMS environment comes from its being a messaging interface with only a few well-defined messages. The most common types of messages are `M_DATA` (for sending raw data), `M_PROTO` (for sending normal commands), and `M_PCPROTO` (for sending high-priority commands). Since STREAMS does not define the content of `M_PROTO` or `M_PCPROTO` messages, it is necessary for modules to agree on a message format if they are to communicate. Apple uses the Transport Provider Interface (TPI) message format for most protocol modules and the **Data Link Provider Interface (DLPI)** for most STREAMS port drivers.

This document assumes familiarity with the STREAMS environment and with the set of STREAMS messages defined by the DLPI specification (*Data Link Provider Interface Specification* by Unix International, OSI Workgroup).

Dynamic Loading

Open Transport supports two methods of dynamically loading STREAMS modules. A STREAMS module may be written as an Apple Shared Library Manager (ASLM) shared library or as a Code Fragment Manager (CFM) code fragment. STREAMS modules written for 68000-family processors must use the ASLM. The CFM is the preferred mechanism for PowerPC modules, but the ASLM may also be used, especially if the module loads C++ classes dynamically.

In this chapter, whenever a STREAMS module or driver is described as exporting a function it means that it exports the function using the named export method of the appropriate DLL. For the ASLM, this means using the `extern` keyword in front of the name of the function in the export file. For the

CFM, this means using the `-export` switch in MPW when linking a shared library.

IMPORTANT

PCI card port drivers for Power Macintosh computers must be written to conform to the new native driver architecture, using the CFM only. Open Transport will get all of the information it needs from the Macintosh Name Registry, described in Chapter 10. ▲

Finding the Driver

For Open Transport to be able to use a port driver, it needs to know that the driver exists. This is accomplished by having a port scanner register the port driver with Open Transport. On Power Macintosh computers with the native driver architecture, Open Transport provides this scanner, and driver writers only need to know how to set up the driver so that it can be found. With other computers, the driver writer may need to provide the port scanner.

Native Port Drivers

Open Transport provides the expert for drivers written for PCI-based Power Macintosh computers with the native driver architecture. For your driver to be automatically located and installed by the Open Transport expert, you must first define and export a `DriverDescription` structure as part of your driver so that your driver is added to the Name Registry. This structure is described in "Driver Description Structure" (page 198).

For Open Transport, the fields of the `DriverDescription` structure must be set as follows:

`driverDescSignature`	Must contain the value `kTheDescriptionSignature`.
`driverDescVersion`	Must contain the value `kInitialDriverDescriptor`.
`driverType.nameInfoStr`	Fill in with the name of the driver. It must be exactly the same name as the module name pointed to by the `streamtab` structure of the driver (in the

qi_minfo->mi_idname field). The driver name may not end in a digit.

driverType.version
: Fill in with the version number of the driver (not the version number of the device, which is stored in the driverDescVersion.revisionID field).

DriverOSRuntimeInfo.driverRuntime
: This field must have the bit kdriverIsUnderExpertControl set.

DriverOSRuntimeInfo.driverName
: This field must contain one of the device names found in OpenTptLinks.h. These include kEnetName, kTokenRingName, kFDDIName, and so on. Remember that this field is a Pascal string, and the equates are for C strings, so you must use code such as "\p" kEnetName to get the desired effect.

DriverOSRuntimeInfo.driverDescReserved[8]
: These are reserved fields and should be initialized to 0.

DriverOSService.service[x].serviceCategory
: At least one of your service categories must be filled in with the category kServiceCategoryopentransport.

DriverOSService.service[x].serviceType
: The service type field is a bit field that tells Open Transport about your device. It has this form:

```
xxxxdddd dddddddd cccccccc xxxxxxTD
```

where the d bits indicate the device type for Open Transport, the c bits indicate Ethernet framing options (the driver's capability bits), the lower 2 bits (TD) state whether the driver is TPI or DLPI, and all other bits are 0 (shown by x). The macro

```
OTPCIServiceType(devType, capabilityBits, isTPI, isDLPI)
```

should be used to create this field. The list of device types available is found in the header file OpenTptLinks.h.

DriverOSService.service[x].serviceVersion
: This field specifies the version of the Open Transport programming interface that your driver supports. It is in the standard NumVersion format (the format of a 4-byte 'vers' resource). Currently, this field should be set to the constant kOTDriverAPIVersion.

Installing the Driver

Once your driver is registered with Open Transport, it is ready for Open Transport to install in a stream. This section describes the installation and loading processes.

Driver Initialization

Any necessary driver initialization should be done by the port scanner before registering the driver. This insures that a device that is not usable does not get registered. For systems using the native driver architecture, Open Transport's port scanner will call `ValidateHardware` before registering your port.

```
OTResult ValidateHardware (RegEntryIDPtr)
```

The parameter passed to the `ValidateHardware` function depends on the port scanner being used. If the driver is able to change the power level of the device, it must use the `ValidateHardware` function, setting the device to either low power or no power.

Open Transport requires that `ValidateHardware` be exported. When this function is called, it should validate that the hardware is correct for the driver and is in good working order. If the function returns `kENOENTErr`, then the hardware is probably not the hardware for the driver and Open Transport will continue scanning for another driver. This is especially important for cards that do not have Open Firmware ROMs, because multiple vendors' drivers may end up with the same name and appear to be usable with each other's hardware.

For information about Mac OS services available to support `ValidateHardware`, see "Driver Initialization and Resource Verification" (page 167).

`ValidateHardware` should return one of the following values:

`kOTNoError`	The hardware is OK. The device will be registered, and the driver may be unloaded from memory.
`kOTPCINoErrorStayLoaded`	The hardware is OK, the device will be registered, and the driver will not be unloaded from memory.

`kENXIOErr`	The hardware is correct for the driver but is not OK. The port will not be registered, and the driver will be unloaded from memory.
`kENOENTErr`	The hardware is probably not correct for the driver. The port will not be registered, and the driver will be unloaded. Open Transport will continue scanning for other drivers that might work with the hardware.
number < 0	Any appropriate error code (such as `kENOMEMErr`). The port will not be registered, and the driver will be unloaded.

If the `ValidateHardware` function is not exported, Open Transport will proceed as if the function returned `kOTNoError`.

Driver Loading

When a service requires the use of your driver, Open Transport will automatically load it and install it into the STREAMS module tables. In order to do this, your module must export a function named either `GetOTInstallInfo` or `GetOT`*xxxxx*`InstallInfo` (where *xxxxx* is the name of the module or driver).

```
install_info* GetOTInstallInfo(void);
```

This function returns the installation information that Open Transport needs to install the driver into the STREAMS tables, using the following data structure:

```
structure install_info
{
structure streamtab*    install_str;
UInt32                  install_flags;
UInt32                  install_sqlvl;
char*                   install_buddy;
void*                   ref_load;
UInt32                  ref_count;
};
```

Field descriptions

`install_str`	This is a pointer to the driver's streamtab structure.
`install_flags`	This contains flags to inform Open Transport of your driver's STREAMS module type. The `install_flags` should

	be set to `kOTModIsDriver | kOTModIsPortDriver` for STREAMS port drivers.
`install_sqlvl`	This flag is set to the type of reentrancy your driver can handle. Possible values are the following:

`SQLVL_QUEUE`	The driver can be entered once from the upper queue and once from the lower queue at the same time.
`SQLVL_QUEUEPAIR`	The driver can be entered from either the upper queue or the lower queue, but not at the same time.
`SQLVL_MODULE`	The driver can be entered only once per port, regardless of which instance of the module is entered.
`SQLVL_GLOBAL`	Among all modules that use `SQLVL_GLOBAL` only one will be entered at a time.

`install_buddy`	This field is currently not support by Open Transport. It should be set to `NULL`.
`ref_load`	This field keeps a load reference to the driver. It should be initialized to 0 and then never touched.
`ref_count`	This field monitors when a driver is first loaded and last unloaded. It should be initialized to 0 and then never touched.

Whenever Open Transport loads your module or driver, and the `ref_count` field of the `install_info` structure is 0, Open Transport will call an optional initialization function exported by the module. This function must be named either `InitStreamModule` or `Init`*xxxxx*`StreamModule` (where *xxxxx* is the name of the module or driver).

```
    Boolean InitStreamModule (void* systemDependent);
```

If `InitStreamModule` returns `false` to Open Transport, then the loading of the module will be aborted and an `ENXIO` error will be returned to the client. Otherwise, the module will be loaded and installed into a stream.

The `systemDependent` parameter is a pointer to the `cookie` value used when registering the port. For drivers loaded using the System registry, its value is `RegEntryIDPtr`.

If the PCI device supports changing power levels, the `InitStreamModule` function should set the power level for normal operation.

Whenever Open Transport removes the last instance of a module or driver from the system, it calls an optional termination function exported by the module. This function must be named either `TerminateStreamModule` or `Terminate`*xxxxx*`StreamModule` (where *xxxxx* is the name of the module or driver).

```
void TerminateStreamModule (void);
```

If the PCI device supports changing power levels, the `TerminateStreamModule` function should set the power level to low power or no power, as appropriate.

Of course, modules and drivers may also use the initialization and termination features of their DLL technology. Both CFM and ASLM allow initialization and termination routines. However, only a call to `InitStreamModule` implies that the module is about to be loaded into a stream. Open Transport often loads a module just to call the `GetOTInstallInfo` information.

All memory allocations that do not use the Open Transport allocation routines (`OTAllocMem` and `OTFreeMem`) or any interrupt-safe allocators supplied by the interrupt subsystem must be performed from within the initialization and termination routines—that is, `PoolAllocateResident` and `PoolDeallocate` may be called only from them.

Once your port driver has been loaded, all communication with it will be through STREAMS messages and the entry points in the `streamtab`.

Note
Native drivers usually require a `DoDriverIO` export. Drivers that only support Open Transport do not need this export, and all references to it in the driver documentation may be safely ignored. ◆

Driver Operation

Once your driver is installed in a stream and opened, it is ready for action. From that point on, the driver will respond to messages according to the interface specifications (TPI or DLPI) that it supports.

Drivers have one additional requirement they must observe. If they are running as a result of a primary interrupt, they must call the `OTEnterInterrupt` function before making any Open Transport calls. They must call `OTLeaveInterrupt` before exiting their current interrupt level, after they have made their final call to any Open Transport routines.

It is strongly suggested that the appropriate Open Transport functions be used for timing services and secondary interrupt services, so they will be most compatible with future versions of Mac OS. Open Transport is also compatible with current non-PCI Macintosh platforms.

The Open Transport secondary interrupt services do not have the same restrictions as some other services, because any memory allocations needed are handled early. This prevents these functions from failing at inconvenient times.

Interrupt-Safe Functions

Open Transport provides many STREAMS services for module and driver writers, but not all of these services may be used at interrupt time.

The following STREAMS functions may be safely called at interrupt time:

`allocb`	`adjmsg`	`copyb`	`copymsg`	`dupb`
`dupmsg`	`esballoc`	`freeb`	`freemsg`	`linkb`
`msgdsize`	`msgpullup`	`pullupmsg`	`rmbv`	`testb`
`unlinkb`	`datamsg`	`OTHERQ`	`RD`	`WR`
`bzero`	`bcopy`	`bcmp`	`putq`	

IMPORTANT

The `putq` function may be used only to put a packet onto its lower (read) queue. No other put operation is allows at interrupt time. In particular, the `canput` function and its variants, as well as the queue enabling and put functions, cannot be called at primary interrupt time. ▲

The following Open Transport functions may be safely called at interrupt time:

`OTCreateDeferredTask`	`OTDestroyDeferredTask`

```
OTScheduleDeferredTask              OTGetClockTimeInSecs

OTGetTimeStamp                      OTSubtractTimeStamps

OTTimeStampInMilliseconds           OTTimeStampInMicroseconds

OTElapsedMilliseconds               OTElapsedMicroseconds

cmn_err             OTAllocMsg              OTAllocMem

OTFreeMem           mi_timer_alloc          mi_timer_free

mi_timer            mi_timer_cancel
```

In addition, all functions described in "Atomic Services" (page 544) may be called at interrupt time.

Secondary Interrupt Services

The functions described in this section are associated with Open Transport's secondary interrupt services.

```
typedef void (*OTProcessProcPtr)(void* contextInfo);
```

This typedef defines the deferred task callback function.

```
long OTCreateDeferredTask (
                     OTProcessProcPtr proc,
                     void *contextInfo);
```

This function creates a cookie (the returned `long` value) that can be used at a later time to schedule the function `proc`. At the time that `proc` is invoked, it will be passed the same `contextInfo` parameter that was passed to the `OTCreateDeferredTask` procedure.

```
void OTScheduleDeferredTask(long dtCookie);
```

This function is used to schedule the deferred procedure corresponding to the `dtCookie` value. It may be called multiple times before the deferred procedure actually being executed, but the deferred procedure will only be run once. Once the deferred procedure has run, subsequent calls to `OTScheduleDeferredTask` will cause it to be scheduled to run again.

```
void OTDestroyDeferredTask(long dtCookie);
```

This function is used to destroy any resources associated with the deferred procedure; it should be called when the procedure is no longer needed.

Timer Services

Open Transport supplies robust timer services that are synchronized with the STREAMS environment and are supported by using special STREAMS messages. The function `mi_timer_alloc` creates one of these special STREAMS messages:

```
mblk_t* mi_timer_alloc(queue_t* targetQueue, size_t size);
```

Calling this function creates a STREAMS timer message of the requested size that is targeted to the specified STREAMS queue. Upper queues must be used as the targets of timer messages because timer messages enter target queues as `M_PCSIG` messages, which can never legitimately arrive from an upper queue but might legitimately arrive from a lower queue.

```
void mi_timer(mblk_t* timerMsg, unsigned long milliSeconds);
```

This function schedules the `timerMsg` (created using `mi_timer_alloc`) to be placed on the target STREAMS queue at a specified future time.

Note
To reset a timer, you need only call `mi_timer` with the new time. There is no need to call `mi_timer_cancel`. ◆

```
void mi_timer_cancel(mbk_t* timerMsg);
```

This function cancels an outstanding timer message. The `timerMsg` message is not destroyed but will no longer be delivered to the target queue. It may be rescheduled by using `mi_timer` at a later time.

```
void mi_timer_free (mblk_t* timerMsg);
```

This function cancels and frees the specified timer message (mi_timer_cancel does not free the message). Never call freeb or freemsg for a timer message.

```
Boolean mi_timer_valid (mblk_t* timerMsg);
```

Timer messages enter the target queue as M_PCSIG messages. Whenever a queue that can receive a timer message receives an M_PCSIG message, it should call mi_timer_valid, passing the M_PCSIG message as a parameter. If the function returns true, then the timer message is valid and should be processed. If the function returns false, then the timer message was either deleted or canceled. In this case, ignore the message and don't free it.

▲ **WARNING**
The mi_timer_valid function may not be called at interrupt time. ▲

```
mblk_t* mi_timer_qswitch
                (mblk_t* timerMsg, queue_t* q, mblk_t* newTimerMsg);
```

This function is called to change the target queue of a timer message. The caller must be in a context that blocks delivery of the timer message to the target queue's put or service routine during the call. For example, the caller must already be in a put or service routine and won't be processing a timer message reentrantly.

The timerMsg parameter is the timer message that is to be moved to the new queue. The q parameter is the new target queue for the timer message. The newTimerMsg parameter is a copy of the timer message that is pointed to by timerMsg. The routine returns a pointer to the timer message that lives on—either timerMsg or newTimerMsg. The other message is freed. If no new message is provided (newTimerMsg is null), but a message is required to do the switch successfully, a null pointer is returned. Both timerMsg and newTimerMsg are copies of the same message. On return, these pointers must be treated as invalid pointers and only the function return pointer can be considered valid.

Atomic Services

Open Transport supplies atomic services that help reduce the need for drivers to disable and enable interrupts.

Note
Don't confuse these services with the DSL atomic services described in Chapter 11. ◆

IMPORTANT
Many atomic services have strict alignment requirements. Be sure to heed the following warnings. The `OTAllocMem` and all STREAMS message blocks are guaranteed to be aligned to 32-bit boundaries. On STREAMS message blocks, this applies to the actual start of the message, not the `b_rptr` field itself, which may not be aligned at all. In 16-bit operations, if the 16 bits cross a 32-bit boundary the atomic function will not work properly. In 32-bit functions, it is important that the variable being operated on be aligned on a 32-bit boundary. ▲

The first set of services atomically sets, clears, or tests a single bit in a byte. The first parameter is a pointer to a single byte, and the second is a bit number from 0 to 7. The functions return the previous value of the bit. Bit 0 corresponds to a mask of 0x01, and bit 7 corresponds to a mask of 0x80.

```
Boolean OTAtomicSetBit      (UInt8* theByte, size_t theBitNo);
Boolean OTAtomicClearBit    (UInt8* theByte, size_t theBitNo);
Boolean OTAtomicTestBit     (UInt8* theByte, size_t theBitNo);
Boolean OTAcquireLock       (UInt8* theByte);
void    OTClearLock         (UInt8* theByte);
```

`OTAcquireLock` is a faster equivalent of `OTAtomicSetBit(theByte, 0)`. It returns `true` if the lock could be acquired (that is, if the bit was flipped from off to on). `OTClearLock` is a macro that just zeroes the byte.

The second set of services atomically add to a 32-, 16-, or 8-bit variable. By using a negative number, they can subtract. The return value is the new value of the variable
as it is when the operation is completed.

```
SInt32 OTAtomicAdd32 (SInt32, SInt32* varToBeAddedTo);
SInt16 OTAtomicAdd16 (SInt16, SInt16* varToBeAddedTo);
SInt8  OTAtomicAdd8  (SInt8,  SInt8*  varToBeAddedTo);
```

The third service is a general compare and swap. It determines if the value at `where` still contains the value `oVal`; if so, it substitutes the value `nVal`. If the compare and swap succeeds, the function returns `true`, otherwise `false`.

```
Boolean OTCompareAndSwap32
                    (UInt32 oVal, UInt32* nVal, UInt32** where);
Boolean OTCompareAndSwap16
                    (UInt16 oVal, UInt16* nVal, UInt16** where);
Boolean OTCompareAndSwap8
                    (UInt8 oVal,  UInt8* nVal,  UInt8** where);
```

The fourth set of services is an atomic last in, first out (LIFO) list. `OTLIFOEnqueue` and `OTLIFODequeue` are self-explanatory. `OTLIFOStealList` lets you remove all of the elements from the LIFO list atomically, so that the elements in the list can be iterated at your leisure by traditional means. `OTLIFOReverseList` is for those who find that LIFO lists are next to useless in networking. Once the `OTLIFOStealList` function has been executed, the result can be passed to `OTLIFOReverseList`, which can be used to flip the list into a first in, first out (FIFO) configuration. The `OTLink` and the `OTLIFO` parameters must both be aligned on 32-bit boundaries. Note that `OTLIFOReverseList` is not atomic.

```
struct OTLink
{
    void*   fNext;
};

struct OTLIFO
{
    void*   fLink;
};

void        OTLIFOEnqueue   (OTLIFO* list, OTLink* toAdd);
OTLink*     OTLIFODequeue   (OTLIFO* list);
OTLink*     OTLIFOStealList (OTLIFO* list);
OTLink*     OTReverseList   (OTLink* firstInList);
```

The last set of services performs enqueueing and dequeueing from a LIFO list. It is used internally in the STREAMS implementation; it is exported so you can use it if it proves useful. If you look at the Open Transport LIFO implementation, it assumes that the structures being linked have their links pointing at the next link, and so on. Unfortunately, STREAMS messages (`msgb` structures) are not linked this way internally (the b_cont field does not point to the b_cont field of the next message block but instead points to the actual message block itself). These two functions let you create a LIFO list where the head pointer of the list points to the actual object, but the next pointer in the object is at some arbitrary offset. It is important that the links and the list itself be aligned on 32-bit boundaries for these functions to work properly.

```
void* OTEnqueue
        (void** list, void* newListHead, size_t offsetOfNextPtr);
void* OTDequeue
        (void** theList, size_t offsetOfNextPtr);
```

Power Services

For those devices that can change their power usage, the STREAMS module must export the entry point OTSetPowerLevel. This lets the system set the device's power level before its driver is installed into a stream.

```
void OTSetPowerLevel(UInt32 powerSelector);
```

In addition, devices that can change their power usage should support the I_OTSetPowerLevel IOCTL call. However, I_OTSetPowerLevel is used only if the driver is already installed into a stream.

Following are the four-byte selectors that can be passed to I_OTSetPowerLevel, with their return values:

'pmn3'	Returns the card's maximum power consumption in microwatts from the 3.3 V supply while in low power mode.
'pmn5'	Returns the card's maximum power consumption in microwatts from the 5 V supply while in low power mode.
'pmx3'	Returns the card's maximum power consumption in microwatts from the 3.3 V supply while in high power mode.
'pmx5'	Returns the card's maximum power consumption in microwatts from the 5 V supply while in high power mode.
'psta'	Returns a value of 1 if the card is in high power mode.
'psup'	Returns a value of 1 if the card supports power control, 0 if it does not.

`'ptog'`	Returns a value of 1 if the card supports switch between high and low power after initialization, 0 if it does not.
`'sphi'`	Sets the card to high power mode. Returns a value of 0 if completed successfully, `OSErr` if not.
`'splo'`	Sets the card to low power mode. Returns a value of 0 if completed successfully, `OSErr` if not.

CSMA/CD Driver

The Open Transport CSMA/CD driver is a STREAMS driver that presents a DLPI to its clients. It is based on Revision 2.0.0 of the DLPI Specification, and is a Style 1 provider, supporting the connectionless mode primitives. Developers who wish to write CSMA/CD drivers that will interoperate with the Open Transport AppleTalk and TCP/IP implementations should use the information given in this section to guide their implementation.

Supported DLPI Primitives

The following DLPI primitives are supported by the Open Transport CSMA/CD driver. The ones marked with a † are not required by either the Appletalk or TCP/IP stacks:

```
DL_BIND_ACK
DL_BIND_REQ
DL_DISABLEMULTI_REQ
DL_ENABLEMULTI_REQ
DL_ERROR_ACK
DL_INFO_ACK
DL_INFO_REQ
DL_OK_ACK
DL_PHYS_ADDR_ACK
DL_PHYS_ADDR_REQ
DL_SUBS_BIND_ACK
DL_SUBS_BIND_REQ
DL_TEST_CON †
DL_TEST_IND †
DL_TEST_REQ †
```

`DL_TEST_RES` †
`DL_UNBIND_REQ`
`DL_UNITDATA_IND`
`DL_UNITDATA_REQ`
`DL_XID_CON` †
`DL_XID_IND` †
`DL_XID_REQ` †
`DL_XID_RES` †

Future versions of the driver will also support these additional primitives:
`DL_GET_STATISTICS_ACK` †
`DL_GET_STATISTICS_REQ` †
`DL_PROMISCOFF_REQ`†
`DL_PROMISCON_REQ`†

Extensions to the DLPI

In addition to supporting the DLPI primitives listed above, the Open Transport CSMA/CD driver includes extensions to support Fast Path mode (described in “Fast Path Mode” (page 557)). This includes the handling of `M_IOCTL` messages with a type of `DL_IOC_HDR_INFO` and special handling of `M_DATA` messages. It also defines several special `M_IOCTL` messages that enable the reception of raw packets and inform the CSMA/CD driver what kind of framing the client expects.

Packet Formats

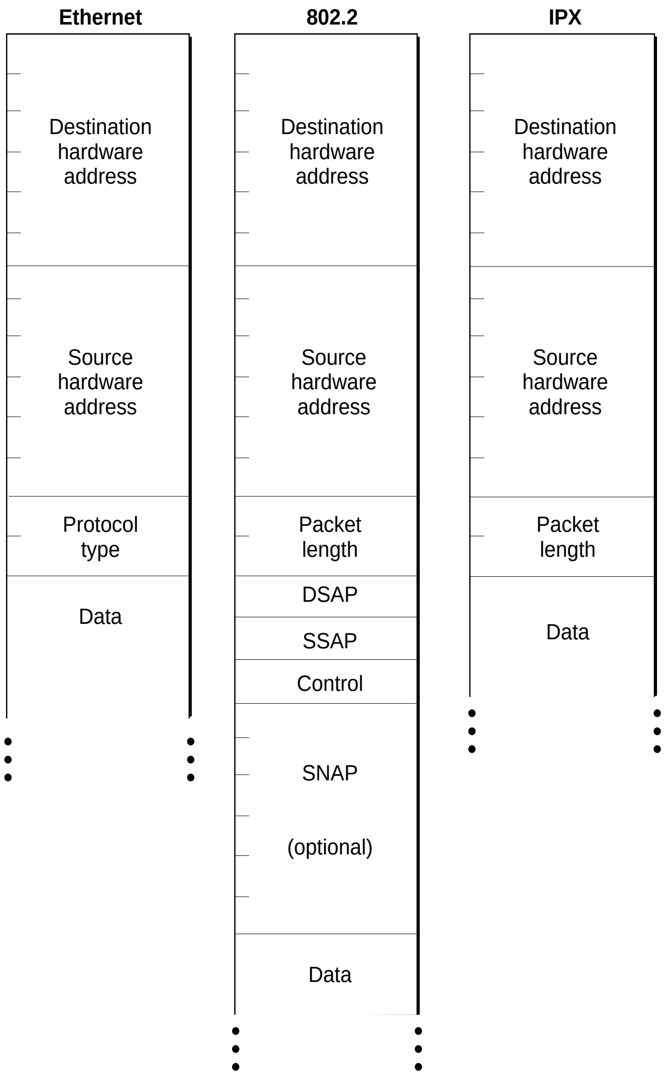
The Open Transport CSMA/CD driver recognizes three packet formats. They are Ethernet, 802.2, and Novell “Raw 802.3,” a version of IPX. The details of the packet format are largely hidden from the client by the driver.

The type of packets the driver will handle is specified at bind time.

In all three packet formats, the first 6 bytes are the destination hardware address, and the next 6 bytes are the source hardware address. The first 6 bytes are followed by a protocol-dependent section, followed by the packet data.

The packet formats that the DSMA/CD driver can handle are diagrammed in Figure 14-1.

Figure 14-1 Packet formats recognized by the CSMA/CD driver



Note
The 802.2 standard is described in *Logical Link Control,* ANSI/IEEE Standard 802.2-1985. ◆

Ethernet Packets

In Ethernet packets, the protocol-dependent section consists of a 2-byte protocol type field. This field has a value in the range 1501 to 65535 (0x5DD to 0xFFFF).

802.2 Packets

In 802.2 packets, the protocol-dependent section consists of a 2-byte length word, a 1-byte destination service access point (DSAP), a 1-byte source service access point (SSAP), a control byte, and an optional 5-byte subnet access protocol (SNAP) field. Thus this section of the packet can be either 5 or 10 bytes long.

Note
The 802.3 specification guarantees that the value of the 2-byte length word will always be less than 1501; therefore it is always possible to differentiate between Ethernet and 802.2 packets by examining the value of this field. ◆

IPX Packets

IPX payloads may be carried in any one of three frames. In addition to Ethernet and 802.2, an IPX packet may be framed in what Novell calls a “Raw 802.3” packet. In this case, the protocol-dependent section consists only of a 2-byte length word. To distinguish these packets from 802.2 packets, Novell specifies that the first 2 bytes of the data section are always set to 0xFF.

Address Formats

Addresses used by the Open Transport CSMA/CD driver consist of two parts—a hardware address and a protocol-dependent field. The hardware address is a 6-byte Ethernet address. A hardware address of all 1s is the broadcast address. If a hardware address is not all 1s but the low bit of the first (left most) byte is set, then the address is a multicast address. The protocol address consists of a 2-byte value called a data link service access point (DLSAP), which corresponds to the DLSAP defined in the DLPI specification. It is optionally followed by a 5-byte SNAP. The protocol address, when present, is appended to the hardware address.

Ethernet

In Ethernet, the DLSAP corresponds to the protocol type field.

802.2

In 802.2 packets, the DLSAP corresponds to either

- The SSAP (in a `DL_BIND_REQ`, `DL_BIND_ACK`, or in the source address field of a `DL_UNITDATA_IND` primitive) or
- The DSAP (in a `DL_UNITDATA_REQ` or in the destination address field of a `DL_UNITDATA_IND` primitive)

If the DLSAP is 0xAA, then it must be followed by a 5-byte SNAP.

IPX

In IPX packets, the DLSAP is always 0x00FF.

Binding

The information passed in a bind request is a function of the type of packets to be handled by this stream—Ethernet, 802.2, or IPX. In all three cases, the `dl_max_conind` field should be set to 0 and the `dl_service_mode` field must be set to the constant `DL_CLDLS`.

Note
The DLPI specification leaves open the possibility that several streams on the same hardware port could be bound to a single DLSAP. This feature is explicitly supported by the Open Transport CSMA/CD driver. If a packet arrives addressed to two or more streams simultaneously, each stream receives a copy of the packet. ◆

Ethernet

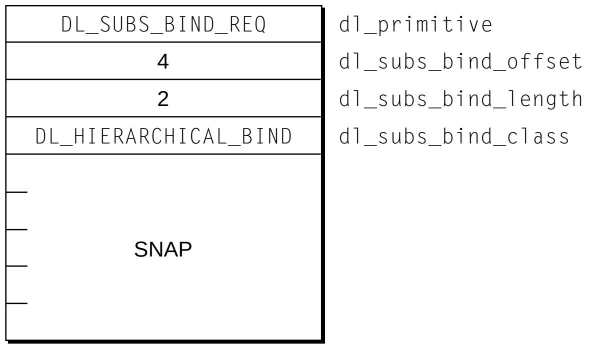
To bind to an Ethernet protocol, the client sends a `DL_BIND_REQ` with the `dl_sap` field set to the protocol type. This is a value in the range 1501 to –65535 (0x5DD to 0xFFFF). The `dl_xidtst_flg` field is ignored.

802.2

To bind to an 802.2 address, the client sends a `DL_BIND_REQ` with the `dl_sap` field set to the SSAP. This is an even value in the range 0 to 254 (0x0 to 0xFE). The `dl_xidtst_flg` field may optionally have either or both of the `DL_AUTO_XID` or `DL_AUTO_TEST` bits set.

If the SSAP is 0xAA, then the client should follow the acknowledgment of the bind with a `DL_SUBS_BIND_REQ` with a 5-byte SNAP. The `dl_subs_bind_class` field should be set to `DL_HIERARCHICAL_BIND`. The message for enabling a SNAP is shown in Figure 14-2.

Figure 14-2 Message for enabling a SNAP

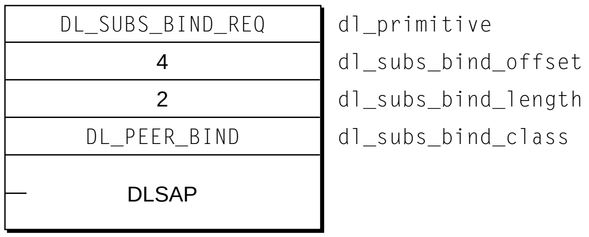


Note
Attempting to perform a hierarchical `subs_bind` operation to any service access point (SAP) value other than 0xAA will cause an error. ◆

After successfully binding to an 802.2 SAP, the client may enable a group SAP by sending a `DL_SUBS_BIND_REQ` with a 2-byte DLSAP containing the group SAP.

Valid group SAPs are odd numbers in the range 1 to 253 (0x1 to 0xFD). In this case, the `dl_subs_bind_class` field should be set to `DL_PEER_BIND`. Note that SAP 255 (0xFF) is the global (broadcast) SAP and is always enabled. The message for enabling a group SAP is shown in Figure 14-3.

Figure 14-3 Message for enabling a group SAP



Note
For a description of group and global SAPs, see ANSI/IEEE Standard 802.2-1985. ◆

As a special case, a client may request that it receive all 802.2 packets that come in. It does so by sending a `DL_SUBS_BIND_REQ` with a 2-byte DLSAP set to 0. The `dl_subs_bind_class` field should be set to `DL_PEER_BIND`.

Note
When sending packets to DLSAP 0xFF, it is ambiguous whether the packet is destined for an 802.2 global SAP or an IPX SAP. The ambiguity is resolved by declaring that only an IPX endpoint can send to another IPX endpoint and an IPX endpoint cannot send to a global SAP. ◆

IPX

To bind to an IPX protocol, the client sends a `DL_BIND_REQ` with the `dl_sap` field set to 255 (0xFF). The `dl_xidtst_flg` field is ignored.

Multicasts

A multicast address may be enabled on a driver with the `DL_ENABMULTI_REQ` message. The value must be a valid multicast address as defined in "Address Formats" (page 551).

Similarly, a multicast address may be disabled on a driver with the `DL_DISABMULTI_REQ` message. The value must be a valid multicast address that was enabled on that particular stream with a prior `DL_ENABMULTI_REQ`.

Sending Packets

Packets are sent with the `DL_UNITDATA_REQ` message. If the destination has the same protocol address as the sender, it is only necessary to supply the hardware address of the destination; otherwise the full address must be used. Note that only a stream bound to the IPX SAP can send to another IPX stream.

To support Fast Path mode, the Open Transport CSMA/CD driver treats `M_DATA` messages as fully formed ("Raw") packets, including all addresses and headers. The only modification made before sending the packet to the hardware is to check for a 0 in the 802.2 length field. If 0 is found, the length field is set to the appropriate value. Support of this feature is optional; see "Fast Path Mode" (page 557) for further information.

Receiving Packets

Incoming packets are passed to the client in `DL_UNITDATA_IND` messages. The `dl_group_address` field is set to 0 if the packet was addressed to a standard Ethernet address. It is set to `keaMulticast` if the packet was addressed to a multicast address and to `keaBroadcast` if the packet was addressed to a broadcast address, where `kaeMulticast` and `kaeBroadcast` are constants (currently 1 and 2, respectively).

The data portion of the message consists of everything following the protocol-dependent section.

Raw Packets

Occasionally, a client may wish to send or receive “Raw” packets—packets with the link and protocol headers attached. To send raw packets, the client merely sends them as `M_DATA` messages, as described in “Fast Path Mode” (page 557).

A client that wishes to receive raw packets may send an `M_IOCTL` message with the `ioc_cmd` field set to `kOTSetRawMode` and its chained data block containing a `UInt32` value. The value can be either `kOTRawRcvOn` or `kOTRawRcvOff`, to turn on or off the reception of raw packets. If the driver supports the delivery of raw packets, it responds with an `M_IOCACK` message; otherwise, with an `M_IOCNAK` message.

Raw packets received will have the `kaeRawPacketBit` set in the `dl_group_address` field of the corresponding `dl_unitdata_ind_t`.

Test and XID Packets

The driver includes support for 802.2 test and XID packets.

If the client requested automatic handling of test or XID packets by setting the `DL_AUTO_TEST` or `DL_AUTO_XID` bits in the `dl_xidtest_flag` field of the bind request when binding to an 802.2 DLSAP, then the driver will respond to incoming test or XID packets without notifying the client. If automatic handling has been requested, the client cannot send test or XID packets.

If the client did not request automatic handling of test or XID packets, then incoming test or XID packets will be passed up to the client as `DL_TEST_IND` or `DL_XID_IND` messages. The client should respond to them with `DL_TEST_RES` or `DL_XID_RES` messages.

If automatic handling has not been requested, the client may send test or XID packets with a `DL_TEST_REQ` or `DL_XID_REQ` message. Any responses are passed back to the client as `DL_TEST_CON` or `DL_XID_CON` messages.

Attempts by non-802.2 streams to send DL_TEST_REQ, DL_XID_REQ, DL_TEST_RES, or DL_XID_RES messages are ignored.

Fast Path Mode

Fast Path is an optional optimization wherein the driver supplies the client with a precomputed packet header for a given destination. The client caches the header and copies it directly into packets addressed to that destination before passing them to the driver. The client first requests a header by sending the driver an M_IOCTL message with its ioc_cmd field set to DL_IOC_HDR_INFO and its chained data block containing the dl_unitdata_req_t structure that the client would normally use to send packets to that particular destination. If the driver does not support fast path, it simply responds with an N_IOCNAK message. STREAMS drivers respond with NAK to any IOCTL they can't handle.

If the driver supports fast path, it responds with an M_IOCACK message with the chained data block containing the precomputed header. In the case of 802.2 packets, the length field of the precomputed header is set to 0. The client prepends the header to outgoing packets and passes them to the driver as M_DATA messages. The driver then sends the packet as is, filling in the 802.2 length field if necessary.

Note
The data block returned in the M_IOCACK should not be modified by the client, and it should always be copied with copyb rather than dupb, since the driver may modify it before sending the packet. ◆

Framing and DL_INFO_REQ

To support the TCP/IP stack available with Open Transport, CSMA/CD drivers must support both Ethernet and 802.2 framing (including full SAP/SNAP binding). Because the DLPI specification does not let a driver support multiple kinds of framing, it is ambiguous in specifying how to fill out the dl_mac_type field of a dl_info_ack_t. Open Transport has specified that the default value of this field should beDL_ETHER. Clients may send an M_IOCTL message with the ioc_cmd field set to kOTSetFramingType and its chained data block containing a UInt32 value with a single bit set. If this value is the constant kOTFraming8022, then subsequent DL_INFO_REQ requests should set the dl_mac_type field to DL_CSMACD. If the value is not that constant, then subsequent DL_INFO_REQ requests should set the dl_mac_type field to DL_ETHER.

IMPORTANT

The only thing the foregoing M_IOCTL message affects is the contents of the DL_INFO_ACK. The framing that is actually used by the driver is specified in the bind. ▲

TokenRing and FDDI Drivers

Open Transport TokenRing and Fiber Distributed Data Interface (FDDI) drivers are identical to the CSMA/CD driver with only 802.2 packets and addressing supported. A hardware multicast in TokenRing is a hardware address with the 2 high-order bits of the lef tmost byte set to 1.

C H A P T E R 1 5

SCSI Drivers

This chapter discusses the requirements for writing native driver code to support SCSI devices on PCI cards in PCI-based Power Macintosh computers.

Macintosh SCSI devices are now supported by SCSI Manager 4.3, an enhanced version of the original Macintosh SCSI Manager. The new capabilities of SCSI Manager 4.3 include

- support for asynchronous SCSI I/O
- support for optional SCSI features such as disconnect and reconnect
- a hardware-independent programming interface that minimizes the SCSI-specific tasks a device driver must perform

The hardware-independence features of SCSI Manager 4.3 mean that the equivalent of SCSI driver code is now a software entity called a **SCSI Interface Module (SIM).** This chapter discusses some of the requirements for writing and loading SIMs in PCI-based Power Macintosh computers.

Inside Macintosh: Devices, described in "Apple Publications" (page 26), contains a full discussion of SCSI Manager 4.3. You should read the material in *Inside Macintosh* first. This chapter covers only the changes from that information for SCSI devices based on PCI cards.

The SCSI Expert

The SCSI expert is supplied by Apple in the firmware of PCI-based Power Macintosh computers. For a discussion of experts, see "Terminology" (page 141).

The SCSI expert is simpler than some other experts and places fewer demands on Open Firmware and the native driver model. A PCI card that wants to register a SIM with the SCSI Manager must place information in the device tree that includes its `name` and `reg` properties. To be recognized by SCSI Manager 4.3 as a SCSI device, the device must have a `device_type` property of `'scsi'`. This is important because it is the primary identifier that causes the SCSI expert to load the SIM. The `device_type` property is generated by the Mac OS startup code and is based on the PCI configuration space parameter `class-code`, which must have a value of `"mass storage"` (01). With the `DriverOSService.service[x].serviceCategory` value of `"blok"`, the `device_type` property completely identifies the SIM code to the SCSI expert.

SIMs for Current Versions of Mac OS

With current versions of Mac OS, you can write a native SIM by using the Mixed Mode Manager and passing universal procedure pointers to the transport (XPT) layer when registering the SIM. Native SIMs should also use `CallUniversalProc` when calling XPT routines.

PCI native SIMs are implemented similarly to other native drivers. The SIM installs a driver in the device tree with a `driver,AAPL,MacOS,PowerPC` property. Like other native drivers, SIMs export a driver description structure. The SCSI expert identifies a SIM by examining the service categories supported in the driver descriptor. SIMs have a `serviceCategory` of type `kServiceCategoryScsiSIM`. A driver supporting this service category should export a function named `LoadSIM` with the following interface:

```
OSErr LoadSIM (RegEntryIDPtr entry);
```

The SCSI expert prepares the code fragment and calls this function after the SCSI transport layer is initialized. In response, the SIM should initialize itself the same way a NuBus SIM would by calling `SCSIRegisterBus`, as described in *Inside Macintosh: Devices.* Any nonzero result returned from `LoadSIM` causes the code fragment to be unloaded. Note that this is a `ProcPtr`-based interface, so you must pass `UniversalProcPtr` structures for all entry points. Those passed back by the XPT will also be `UniversalProcPtr` structures so native code should use `CallUniversalProc` when calling XPT layer procedures from the `SIMInitRecord`.

An typical PCI-based SIM descriptor is shown in Listing 15-1.

Listing 15-1 SIM descriptor

```
DriverDescription TheDriverDescription =
{
    // signature information
        kTheDescriptionSignature,
        kInitialDriverDescriptor,
    // type info
        "\pFor Rent                        ",
        1,0,0,0,                             // major, minor, stage, rev
    // OS runtime info
        kDriverIsUnderExpertControl,
        "\p.MySCSISIM                       ",
        0,0,0,0,0,0,0,0,                     // reserve 8 longs
    // OS service info
        1,                                   // number of service
categories
        kServiceCategoryScsiSIM,
        0,
        1,0,0,0                              // major, minor, stage, rev
};
```

For the Startup Disk control panel to be able to select a boot device from a SIM correctly, the `SCSIBusInquiry` fields `scsiHBAslotNumber` and `scsiSIMsRsrcID` must uniquely identify the SIM from other SIMs and PCI cards. Each SIM should identify itself when registering with the system by placing a `RegEntryID` value in the `SIMInitInfo` parameter block. The XPT layer will calculate unique values for the `SCSIBusInquiry` fields and supply them to the `SIMInit` routine. From then on the SIM must return these values from `SCSIBusInquiry`. Three new fields—`simSlotNumber`, `simSRsrcID`, and `simRegEntry`—have been defined in the `SIMInitInfo` parameter block to hold these values. The new parameter block is defined as follows:

```
    UInt8                       *SIMstaticPtr;
    long                        staticSize;
    SIMInitUPP                  SIMInit;
    SIMActionUPP                SIMAction;
    SCSIInterruptUPP            SIM_ISR;
    SCSIInterruptUPP            SIMInterruptPoll;
    SIMActionUPP                NewOldCall;
    UInt16                      ioPBSize;
    Boolean                     oldCallCapable;
    UInt8                       simInfoUnused1;
    long                        simInternalUse;
    SCSIUPP                     XPT_ISR;
    SCSIUPP                     EnteringSIM;
    SCSIUPP                     ExitingSIM;
    SCSIMakeCallbackUPP         MakeCallback;
    UInt16                      busID;
```

```
UInt8                   simSlotNumber;          // output
UInt8                   simSRsrcID;             // output
RegEntryIDPtr           simRegEntry;            // input
```

Future Compatibility

The current SCSI Manager 4.3 interface is not guaranteed to be compatible with future Mac OS releases. At this time the SIM architecture is not fully defined and may be subject to change. However, it is possible to write a fully native SIM by passing universal procedure pointers to the XPT layer for the SIM's entry points and by using `CallUniversalProc` in native code to call the XPT's entry points. This approach is outlined in "SIMs for Current Versions of Mac OS" (page 561). Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software*, listed in "Apple Publications" (page 26).

It is also possible to reduce the effort required to become compatible with future releases of Mac OS by following the rules set forth for other drivers in Chapter 8, "Writing Native Drivers." Primarily, you should limit communication with Mac OS to the calls documented in Chapter 11, "Driver Services Library."

SCSI Device Power Management

Supporting power management in a SCSI driver unavoidably violates some of the guidelines set forth in "Card Power Controls" (page 469). This section discusses some of the issues and potential solutions.

At a minimum, SCSI storage device drivers should support driver gestalt as defined in "Driver Gestalt" (page 221). They should respond positively to the `'lpwr'` gestalt selector. Supporting driver gestalt mandates that the driver support `csCode=70` for getting and setting the low power state (for spindle motor control, in most cases). The currently defined power modes are Active, Standby, Idle, and Sleep.

If a driver does not have to support multiple platforms (such as both Power Macintosh and PowerBook computers) and chooses to rely on the Power Manager's internal timing semaphores, it should implement the following processes:

- Install code in the Power Manager's HD Spindown, Sleep, and State Notification queues.
- Make an `UpdateSystemActivity` call to notify the Power Manager of activity on the driver's associated device.

When these processes are implemented, drivers registered with the Power Manager will not be ordered to enter a low power mode until all devices have been idle for a period of time set by the user. However, no individual device control will be available and more work will be required to make the driver compatible with future releases of Mac OS.

To be compatible with both Power Macintosh and PowerBook computers, or to simply provide a more elegant solution to the user, the driver should maintain an internal timer specifically for the device it administers. If multiple devices are managed by a single driver, multiple timers should be managed as well. To provide this level of support, the following must be implemented:

- Make an `UpdateSystemActivity` call to notify the Power Manager of activity on the driver's associated device. This is required by the Power Manager to track idle time for system sleep correctly.
- Install code in the Power Manager State Notification queue requesting notification of spindown enable and disable changes, changes to the user-defined timeout period, and changes to the hard disk power state.
- Keep an internal timer in the driver and provide some method to update the timer and invoke low power modes when appropriate. A VBL or Time Manager task may be used.

Drivers should not install code into the HD Spindown queue in this implementation. However, if the driver supports the main internal storage device on a PowerBook computer and requires device preparation before power is removed, Sleep and Wake and HD Spindown queue elements should be implemented.

With either Power Macintosh or PowerBook platforms, any access to a driver's device or any driver request that requires the device to be at full power should cause the driver to wake the device before servicing that request. A control call to resume full power must be supported, but such a call is not required to wake the device.

Note
Gestalt checks for the presence of the Power Manager should be made to decide whether to implement a low power solution upon a driver open or acknowledge request and to determine what kind of support is appropriate. ◆

The current Power Manager implementation supports a mixed environment where some clients are dependent on the Power Manager's internal timing semaphore and others are self-sufficient. Drives supported by driver-based timers will spin down on a drive-by-drive basis. The internal timer will still trigger a spindown of those drives that rely on the Power Manager's timing facilities. It would be wise in either implementation to respond intelligently to requests to enter a power mode that is already present.

Appendixes

The following appendixes contain information that supplements the information in the previous chapters:

- Appendix A, "Big-Endian and Little-Endian Addressing," discusses the theory and problems of handling mixed-endian formats.
- Appendix B, "Graphic Memory Formats," describes the ways that graphic information and video frames are stored in PCI-based Power Macintosh computers.
- Appendix C, "PCI Header Files," describes the PCI header files and lists all the routines and data structures documented in this book.
- Appendix D, "Abbreviations," lists the abbreviations and acronyms used in this book.

A P P E N D I X   A

Big-Endian and Little-Endian Addressing

PCI-based Power Macintosh computers are **mixed-endian** because they support both big-endian and little-endian data formats. This appendix presents solutions to some of the problems that the computers encounter because they support both formats.

Although the natural addressing mode of the PowerPC microprocessor is big-endian, PCI-based Power Macintosh computers support little-endian addressing for several reasons:

- because the PCI bus is little-endian
- so that they are compatible with expansion cards that store data in little-endian format
- so that they can run operating systems (such as Windows NT) that require the underlying hardware to operate as if it were little-endian

This appendix first discusses the theory of big-endian and little-endian addressing and then examines how PCI-based Power Macintosh computers deal with the resulting problems and issues.

Note
The terms *big-endian* and *little-endian* come from Jonathan Swift's eighteenth-century satire *Gulliver's Travels.* The subjects of the empire of Blefuscu were divided into two factions: those who ate eggs starting from the big end and those who ate eggs starting from the little end. ◆

Endian Theory

To give a concrete example around which to discuss endian format issues, consider writing code for a system that contains a DBDMA-like controller. The DMA code includes a descriptor format whose C definition might be as follows:

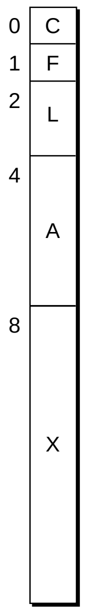
```
struct {
    UInt8       C;      // "command" byte
    UInt8       F;      // "flags"
    UInt16      L;      // "length" (count)
    UInt32      A;      // "address"
    UInt64      X;      // "field64"
        }       DMA_Descriptor;
```

A compiler would assign offsets to the fields of the descriptor as follows:

```
C       0
F       1
L       2
A       4
X       8
```

Consider the diagram in Figure A-1, which presents the layout of the descriptor in a format that is neither big-endian nor little-endian. In Figure A-1, the numbers represent byte offsets to the descriptor's fields.

Figure A-1 Neutral descriptor layout



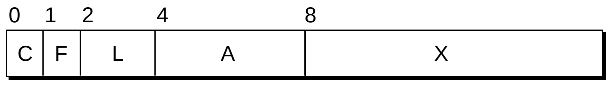
In Figure A-1 the byte offsets are associated with the “beginning” of each field. As discussed in the next sections, the primary difference between big-endian and little-endian addressing has to do with what is defined as the “beginning” of a field.

Big-Endian Addressing

Figure A-2 shows what happens when the diagram in Figure A-1 is rotated counterclockwise.

Note
In Figure A-2 and Figure A-3, the organization of memory is shown with the more significant bytes to the left and the less significant bytes to the right. This is consistent with standard numerical notation and most computer system documentation. Likewise, all bit-field and byte-field designations reference the most significant bit or byte number of the field first. ◆

Figure A-2 Big-endian descriptor layout



The diagram in Figure A-2 shows how a big-endian processor or memory system would organize the sample descriptor. In a big-endian system, physical memory is organized with the address of each byte increasing from most significant to least significant.

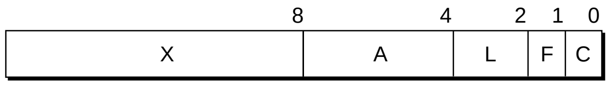
Endian order makes no difference for single-byte values. However, with multibyte values, the endian order determines the order in which bytes are addressed. As noted above, multibyte fields are interpreted with more significant bytes to the left and less significant bytes to the right. This means that the address of the most significant byte of the address field A is 4, while byte 7 corresponds to the least significant byte of A.

Bit ordering in a strictly big-endian architecture should naturally follow the ordering of bytes; that is, the most significant bit should be bit 0. This is true of PowerPC addressing. All bit numbering in this appendix follows the byte order, so the first bit designated in big-endian addressing (the most significant bit) has the lowest bit number.

Little-Endian Addressing

Figure A-3 shows what happens when the diagram in Figure A-1 (page 571) is rotated clockwise.

Figure A-3 Little-endian descriptor layout



This diagram in Figure A-3 shows how a little-endian system would organize the descriptor. Notice which bytes constitute the "beginning" of each field. Instead of referring to the most significant byte of a field, the offsets refer to the least significant byte of each field. Hence, in this example, byte 4 refers to the least significant byte of the A field, while byte 7 refers to the most significant byte.

Bit numbering in a little-endian architecture naturally follows that of byte ordering; that is, bit 0 represents the least significant bit of a field. Thus, in little-endian bit field designations, the first bit shown (the most significant) has the highest bit number.

Scalar Accesses

If all accesses to a data structure were done with read and write actions that transferred a whole field at a time, a program could not determine whether it was executing on a big-endian or little-endian system. For example, a word-sized access to field A in Figure A-1 (page 571) would always get the correct value.

Suppose that the code shown in Listing A-1 is used to initialize the descriptor shown in Figure A-1. The field values chosen in Listing A-1 are encoded: the first nibble gives the size of the field, and the other nibbles represent the byte offsets of each byte, assuming big-endian ordering.

Listing A-1 Field value initializer

```
DMA_Descriptor aDescr;
aDescr.C = 0x10;
aDescr.F = 0x11;
aDescr.L = 0x2223;
aDescr.A = 0x44454647;
aDescr.X = 0x88898A8B8C8D8E8F;
```

In Figure A-1, all accesses to field `aDescr.L` would yield identical results on either a big-endian or little-endian system, so it would normally be impossible to tell whether the system was big-endian or little-endian. However, certain code can detect the order of byte significance relative to the address of the fields initialized by the code shown in Listing A-1 and can thus tell whether the system addresses data in big-endian or little-endian mode. An example is shown in Listing A-2.

Listing A-2 Endian mode determination code

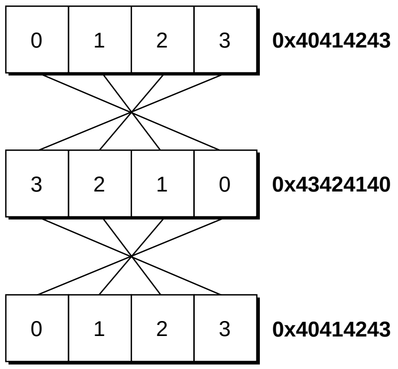
```
union {
        half    H;
        byte    B[2];
        }       halfTrick;
    halfTrick ht;
    ht.H = aDescr.L;
    if( ht.B[0] == 0x22 )
        printf( "I'm on a big-endian system" );
else
        printf( "I'm on a little-endian system" );
```

Address Invariance and Byte Swapping

Address invariance (also called *byte address consistency*) guarantees that individual bytes are mapped across a data bridge according to their address (or byte lane number); the address of a byte is kept the same on both sides of the bridge.

For example, the little-endian NuBus maintains address invariance when passing data between the big-endian Macintosh II computer and an expansion card. To keep track of data movement, bytes are channeled into **byte lanes.** Thus, byte lane 0 of the Macintosh processor bus is mapped to byte lane 0 of NuBus, and so on. But when a 32-bit word passes to NuBus, the bytes are changed in significance by a process called **byte swapping.** The expansion card undoes the byte swap on its side of NuBus, so that data in memory on a card is organized exactly the same way it is on the Macintosh side. The diagram in Figure A-4 shows how data is mapped from the Macintosh II system across NuBus onto an expansion card.

Figure A-4 Byte swapping in NuBus



Note
Byte-swapping is like parity. An even number of byte swaps produces the original ordering. ◆

Mixed-Endian Systems

To use the PCI bus and achieve compatibility with a wide range of expansion card designs, PCI-based Power Macintosh computers are forced to be mixed-endian. This section discusses some of the issues that result from mixed-endian system design.

Transmitting Addresses

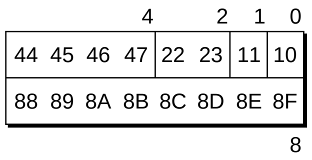
In PCI-based Power Macintosh computers, addresses never require byte swapping. They are written and read as whole quantities and are passed directly across PCI bridges without byte swapping. However, some transformations may be required when transporting addresses across a bridge—for example, to encode byte lanes and transfer sizes. Addresses may also be altered by logical operations, as described in "Address Swizzling" (page 578).

Byte-Swapping Issues

Byte swapping of data is a natural consequence of address invariance. It occurs when data in one endian format is read by a system that uses the other endian format. For example, suppose the DMA descriptor values initialized by the code shown in Listing A-1 (page 573) are generated by a little-endian system and saved to disk. The data is then read from the disk by a big-endian system.

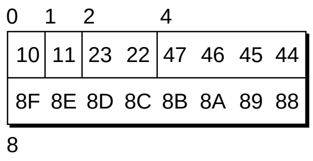
Assume that the data is written to disk in byte-address order, and that the disk memory is formatted in an 8-byte wide configuration. The little-endian disk memory image would look like Figure A-5.

Figure A-5 Little-endian memory image



When read by a big-endian system in byte-address order, the data would be stored in memory as shown in Figure A-6.

Figure A-6 Big-endian memory image

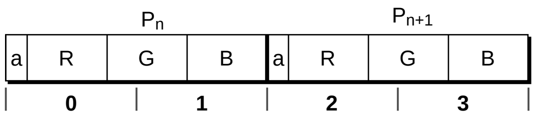


Notice that the byte offsets of each field are still correct. However, the data within each field has been swapped. If field `aDescr.A` was read with a little-endian word loading process, the data in memory would be 0x47464544, even though the original data was written as 0x44454647.

Byte Swapping and Frame Buffers

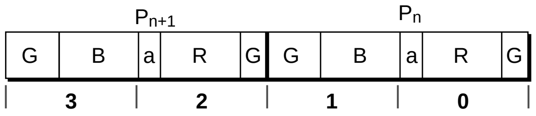
Another example of byte swapping is what happens to multibyte pixels in a frame buffer. Macintosh software is compatible with several multibyte pixel formats, of which 16-bit pixels provide a good example of the effects of byte swapping. The Macintosh 16-bit RGB format interprets a half word as consisting of a 1-bit alpha value followed by three 5-bit red, green, and blue color components. The diagram in Figure A-7 shows how these pixels are packed into a word in big-endian memory.

Figure A-7 Big-endian RGB 16-bit pixel format



When this data is moved across the little-endian PCI bus, data swapping makes the data appear as shown in Figure A-8.

Figure A-8 Little-endian RGB 16-bit pixel format



Notice two effects of the byte swapping process:

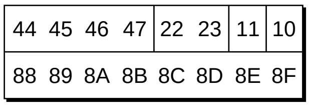
- The relative location of the pixels is correct for the little-endian PCI; this is a direct consequence of maintaining address invariance.
- The data within the pixels has been partly rearranged. For example, the green component has been split into two pieces because it spans a byte boundary.

Address Swizzling

It is possible to make it appear that memory is organized in little-endian format, even though it is maintained by a microprocessor that is inherently big-endian, such as the PowerPC processor. This effect is desirable, for example, when Windows NT runs on a PCI-based Power Macintosh computer, because Windows NT requires memory to appear to be little-endian. It can be achieved by changing addresses without altering the layout of data in memory, a technique called **address swizzling.**

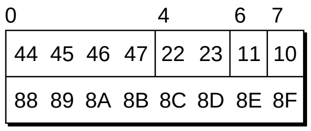
For example, refer to the DMA descriptor values initialized by the code shown in Listing A-1 (page 573). Little-endian software expects the descriptor to be arranged in memory as shown in Figure A-9.

Figure A-9 Little-endian descriptor in memory



A big-endian processor can maintain the memory image shown in Figure A-9 by addressing it with big-endian byte lane assignments, as shown in Figure A-10. If a little-endian processor were maintaining the same image, it would assign byte lanes as shown in Figure A-5 (page 576).

Figure A-10 Little-endian descriptor with big-endian addresses



Within fields, the byte ordering of the data image shown in Figure A-10 is correct, but the data addresses have been swizzled. For example, the field `aDescr.C` that is stored in byte lane 0 in the little-endian format shown in Figure A-5 (page 576) is now stored in byte lane 7 in Figure A-10.

Address swizzling is one technique by which the PowerPC processor provides little-endian addressing support. It is described more fully in "Little-Endian Processing Mode" (page 579).

PowerPC Little-Endian Support

PowerPC microprocessors, which normally address data in big-endian format, provide two separate mechanisms to support little-endian and mixed-endian systems:

- byte-reversed load and store instructions
- little-endian processing mode

These mechanisms are discussed in this section.

Byte-Reversed Load and Store Instructions

The PowerPC instruction set includes a class of load and store instructions that perform byte swapping based on the size of the data transferred. For example, the load word byte reversed indexed (`lwbrx`) instruction swaps a 4-byte value. The primary purpose of instructions such as `lwbrx` is to allow efficient access to data in little-endian format, without additional byte-swapping.

For an example, refer to the big-endian DMA descriptor value shown in Figure A-6. If a program uses a PowerPC `lwbrx` instruction to access field `aDescr.A`, it reads the value 0x44454647, which is the correct data in little-endian format.

Byte-reversed load and store instructions require more code than other load and store instructions, because they exist only in indexed form without update forms. Either addresses of fields within data structures must be explicitly calculated, or field offsets must be loaded into a register. Also, there is currently no C compiler mechanism available to generate these instructions.

Little-Endian Processing Mode

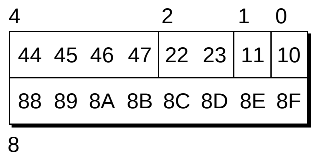
The PowerPC microprocessor supports a little-endian processing mode, in which addresses are swizzled when they are used to access memory. The swizzle applies an XOR operation to the low-order 3 bits of an address with a constant that depends upon the size of the data being loaded or stored. Word load and store actions use a value of 0b100, halves use 0b110, and bytes use 0b111. The resulting addresses are used to make memory references to a big-endian memory system.

Note
The PowerPC's effective address is not modified, only the interpretation used to access memory. For example, the update forms of load and store instructions alter the base register with the same value, regardless of the current endian mode. Thus, the address swizzle is completely transparent to software. ◆

Notice that the address swizzle in little-endian processing mode changes only the lower 3 bits. The number of address bits swizzled depends upon the maximum scalar data type that can be accessed by the system; it does not depend upon the width of the processor's data path. In the case of PowerPC processor, the longest scalar is a double word—hence, swizzling 3 bits suffices to transform any address.

By swizzling the offsets in the big-endian DMA descriptor value shown in Figure A-10 (page 578), little-endian processing mode produces a new set of offsets. For example, the processor applies the calculation 0b000 `XOR` 0b100 to the 0 offset for the word field `aDescr.A`, producing the offset 0b100, or 4. Software can read the correct value of 0x44454647 at that offset. The result is that the whole descriptor appears to have the structure shown in Figure A-11.

Figure A-11 Descriptor swizzled by little-endian processing mode



Note
PowerPC little-endian mode does not support misaligned data accesses. Access to misaligned data must be done by code sequences or subroutines. As is the case with byte-reversed load and store instructions, there is currently no compiler support for handling misaligned data. ◆

Graphic Memory Formats

This appendix describes the various formats in which pixel information is stored in frame buffers in PCI-based Power Macintosh computers. It also includes information about transforming pixel information to convert it from big-endian to little-endian format and vice versa. For information about data formats, see Appendix A, "Big-Endian and Little-Endian Addressing."

The drawings in this appendix that illustrate pixel formats are presented in three parts:

- The top diagram (denoted by BIG) shows the pixel's big-endian format, with the byte lanes numbered in big-endian order.
- The middle diagram (denoted by GIB) shows the pixel value as it appears on the PCI bus, byte swapped to fulfill the PCI bridge's address invariance. This diagram shows the little-endian PCI byte lane numbering.
- The bottom diagram (denoted by LITTLE) shows the little-endian format, with the byte lanes numbered in little-endian order.

Note
All pixel formats shown in this appendix conform to the *PCI Multimedia Design Guide,* listed in "Other Publications" (page 26). ◆

RGB Pixel Formats

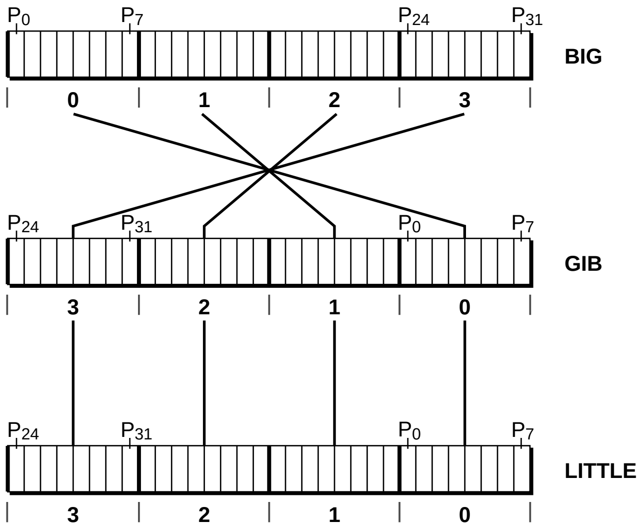
The following sections describe the red-green-blue (RGB) pixel formats that are directly supported by QuickDraw in Mac OS. Where the formats are affected by endian formatting, the BIG, GIB and LITTLE formats are shown.

1, 2, 4, and 8 Bits Per Pixel

With pixel formats 1 byte long or less, no pixel transformation is required, because the bridge's address-invariant byte swapping does not affect data below the byte level. However, it is important to recognize that PCI-based Power Macintosh computers assume that pixels are packed into bytes in left-to-right order. For example, in 1-bit mode the most significant bit of a byte is the leftmost visible pixel on the screen. This is consistent with existing VGA pixel formats.

Figure B-1 shows 1-bit-per-pixel mode. The 2-bit, 4-bit, and 8-bit cases are similar.

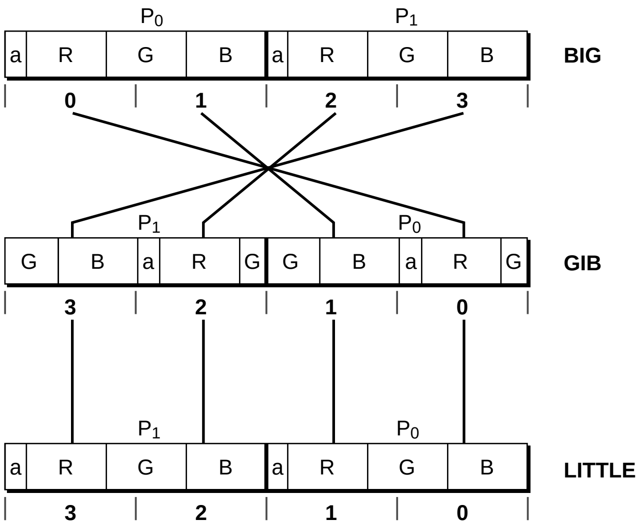
Figure B-1 1-bit-per-pixel formats



16 Bits Per Pixel

16-bit pixel encoding includes a 1-bit alpha value and three 5-bit red, green, and blue color components, as shown in Figure B-2.

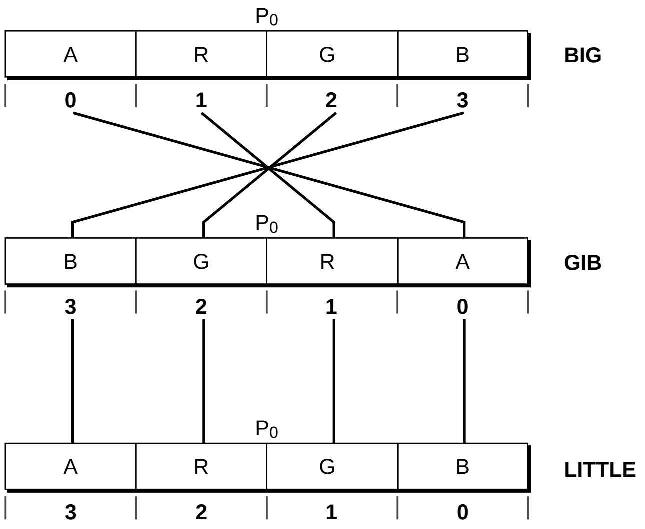
Figure B-2 16-bits-per-pixel formats



24 and 32 Bits Per Pixel

The format of 24- and 32-bit pixels is shown in Figure B-3. In 24-bit mode, the data value of the alpha byte is undefined; however, space is always reserved for it. The 24-bit and 32-bit pixels are always contained within 32-bit words.

Figure B-3 24- and 32-bits-per-pixel formats

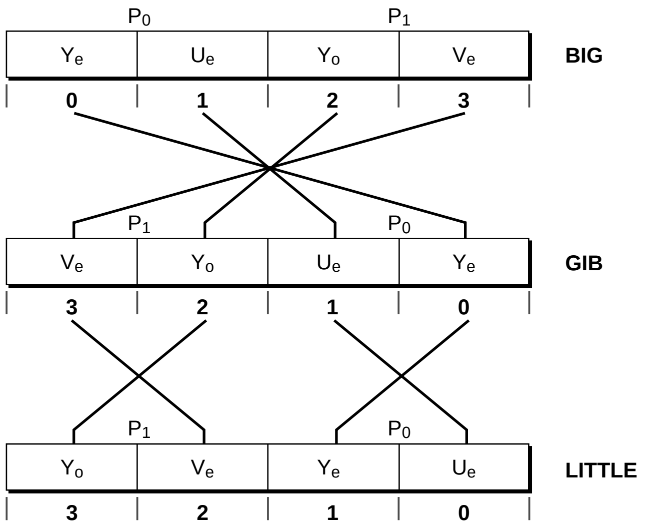


YUV Pixel Formats

YUV pixel formats are typically generated by video input hardware from video cameras, videocassette recorders, and so on; they are not normally generated by software. Although there are various YUV formats possible, determined by the ratio and size of luminance samples (Y) and chroma (U and V) values, PCI-based Power Macintosh computers support only the 4-2-2 format. This format includes two 8-bit Y samples for each pair of 8-bit U and V samples. While 2 pixels (even-odd pairs) are packed into a 32-bit word, each pixel can be thought of as being composed of a luminance component (Y) and a chroma component (U or V) packed into 16-bit values.

The transformations of YUV pixels across a PCI bridge from BIG to GIB format are similar to those of 16-bit pixels. Figure B-4 shows the YUV 4-2-2 pixel formats. As is the case with 16-bit pixels, the pixels in YUV GIB format are in the correct positions but the bytes within each pixel have been swapped.

Figure B-4 YUV pixel formats



Definitions of Pixel Formats in C

Another way to describe the pixel formats in PCI-based Power Macintosh computers is by C struct definitions. The bit packing and bit ordering of packed bit structure fields in C match the endian formats of the target architecture.

Big-endian C compilers pack bits from left to right, while little-endian C compilers pack the bits from right to left. Hence different data structures must be used to describe a given pixel format, depending upon whether the target code is big-endian or little-endian.

Listing B-1 shows how the pixel formats described in this appendix can be defined in C for big-endian and little-endian bit ordering.

Listing B-1 C structures for pixel formats

```
typedef struct {                /* big-endian pixel formats */
    u_int       alpha:1;
    u_int       red:5;
    u_int       green:5;
    u_int       blue:5;
    } RGB_15_alpha;

typedef struct {
    u_int       alpha:8;
    u_int       red:8;
    u_int       green:8;
    u_int       blue:8;
    } RGB_24_alpha;

typedef struct {                /* little-endian pixel formats */
    u_int       blue:5;
    u_int       green:5;
    u_int       red:5;
    u_int       alpha:1;
    } RGB_15_alpha;

typedef struct {
    u_int       blue:8;
    u_int       green:8;
    u_int       red:8;
    u_int       alpha:8;
    } RGB_24_alpha;
```

PCI Header Files

Apple supplies a large number of C-language header files of interest to Macintosh developers. They include interfaces to both Mac OS system software and ROM-based Macintosh startup firmware.

Among these header files are those you need to compile drivers and other PCI-related software for the second generation of Power Macintosh computers. Table C-1 lists them and gives references to the sections of this book where each file's content is discussed.

Table C-1 Header files for Macintosh PCI development

File name	Book reference
`Devices.h`	Chapter 8, "Writing Native Drivers"
`DriverServices.h`	Chapter 11, "Driver Services Library"
`DriverGestalt.h`	"Driver Gestalt" (page 221)
`Interrupts.h`	"Interrupt Management" (page 381)
`Kernel.h`	Chapter 11, "Driver Services Library"
`NameRegistry.h`	Chapter 10, "Name Registry"
`PCI.h`	Chapter 12, "Expansion Bus Manager"
`Video.h`	Chapter 13, "Graphics Drivers"

Table C-2 lists the functions and data structures that the header files listed in Table C-1 support. For each one it gives the name of the supporting file and the page number in this book where the function or data structure is documented.

Table C-2 PCI-related functions and data structures

Function or data structure	Header file	Page
`AbsoluteDeltaToDuration`	`DriverServices.h`	422
`AbsoluteDeltaToNanoseconds`	`DriverServices.h`	422
`AbsoluteTime`	`DriverServices.h`	419
`AbsoluteToDuration`	`DriverServices.h`	421
`AbsoluteToNanoseconds`	`DriverServices.h`	421
`AddAbsoluteToAbsolute`	`DriverServices.h`	421
`AddAtomic`	`DriverServices.h`	427
`AddDurationToAbsolute`	`DriverServices.h`	421
`AddNanosecondsToAbsolute`	`DriverServices.h`	421
`BitAndAtomic`	`DriverServices.h`	427
`BitOrAtomic`	`DriverServices.h`	427
`BitXorAtomic`	`DriverServices.h`	427
`BlockCopy`	`DriverServices.h`	379
`CallSecondaryInterruptHandler2`	`Kernel.h`	412
`CancelTimer`	`Kernel.h`	425
`CDDeviceCharacteristics`	`DriverGestalt.h`	241
`ChangeInterruptSetOptions`	`Interrupts.h`	406
`CheckpointIO`	`Kernel.h`	366
`CompareAndSwap`	`DriverServices.h`	427
`CreateInterruptSet`	`Interrupts.h`	403
`CreateSoftwareInterrupt`	`Kernel.h`	407
`CStrCat`	`DriverServices.h`	431

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
CStrCmp	DriverServices.h	432
CStrCopy	DriverServices.h	430
CStrLen	DriverServices.h	433
CStrNCat	DriverServices.h	431
CStrNCmp	DriverServices.h	432
CStrNCopy	DriverServices.h	431
CStrToPStr	DriverServices.h	433
CurrentExecutionLevel	Kernel.h	347
CurrentTaskID	Kernel.h	407
DecrementAtomic	DriverServices.h	427
DelayFor	Kernel.h	424
DelayForHardware	Kernel.h	424
DeleteSoftwareInterrupt	Kernel.h	409
DeviceProbe	DriverServices.h	171
DriverDescription	Devices.h	198
DriverFinalInfo	Devices.h	206
DriverGestaltBootResponse	DriverGestalt.h	230
DriverGestaltDevTResponse	DriverGestalt.h	231
DriverGestaltIntfResponse	DriverGestalt.h	231
DriverGestaltIsOn	Devices.h	222
DriverGestaltOff	Devices.h	222
DriverGestaltOn	Devices.h	222
DriverGestaltParam	DriverGestalt.h	226
DriverGestaltSyncResponse	DriverGestalt.h	230
DriverGestaltWideResponse	DriverGestalt.h	233

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
DriverInitInfo	Devices.h	206
DriverOSRuntime	Devices.h	201
DriverOSService	Devices.h	202
DriverServiceInfo	Devices.h	203
DriverType	Devices.h	200
DurationToAbsolute	DriverServices.h	421
DurationToNanoseconds	DriverServices.h	421
ExpMgrConfigReadByte	PCI.h	461
ExpMgrConfigReadLong	PCI.h	463
ExpMgrConfigReadWord	PCI.h	462
ExpMgrConfigWriteByte	PCI.h	464
ExpMgrConfigWriteLong	PCI.h	466
ExpMgrConfigWriteWord	PCI.h	465
ExpMgrInterruptAcknowledgeReadByte	PCI.h	467
ExpMgrInterruptAcknowledgeReadLong	PCI.h	468
ExpMgrInterruptAcknowledgeReadWord	PCI.h	467
ExpMgrIOReadByte	PCI.h	455
ExpMgrIOReadLong	PCI.h	457
ExpMgrIOReadWord	PCI.h	456
ExpMgrIOWriteByte	PCI.h	458
ExpMgrIOWriteLong	PCI.h	460
ExpMgrIOWriteWord	PCI.h	459
ExpMgrSpecialCycleBroadcastLong	PCI.h	468
ExpMgrSpecialCycleWriteLong	PCI.h	469
FindDriverCandidates	Devices.h	252

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
`FindDriversForDevice`	`Devices.h`	255
`FlushProcessorCache`	`DriverServices.h`	374
`GetDataCacheLineSize`	`DriverServices.h`	368
`GetDriverDiskFragment`	`Devices.h`	251
`GetDriverForDevice`	`Devices.h`	257
`GetDriverInformation`	`Devices.h`	271
`GetDriverMemoryFragment`	`Devices.h`	250
`GetInterruptFunctions`	`Interrupts.h`	402
`GetInterruptSetOptions`	`Interrupts.h`	405
`GetIOCommandInfo`	`DriverServices.h`	210
`GetLogicalPageSize`	`DriverServices.h`	368
`GetPageInformation`	`Kernel.h`	369
`GetTimeBaseInfo`	`DriverServices.h`	417
`HigherDriverVersion`	`Devices.h`	269
`HighestUnitNumber`	`Devices.h`	273
`IncrementAtomic`	`DriverServices.h`	427
`InstallDriverForDevice`	`Devices.h`	268
`InstallDriverFromDisk`	`Devices.h`	262
`InstallDriverFromFile`	`Devices.h`	265
`InstallDriverFromFragment`	`Devices.h`	261
`InstallDriverFromMemory`	`Devices.h`	266
`InstallInterruptFunctions`	`Interrupts.h`	401
`InterruptDisabler`	`Interrupts.h`	399
`InterruptEnabler`	`Interrupts.h`	399
`InterruptHandler`	`Interrupts.h`	397

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
`InterruptSetMember`	`Interrupts.h`	396
`IOCommandIsComplete`	`DriverServices.h`	192
`IOPreparationTable`	`Kernel.h`	354
`LookupDrivers`	`Devices.h`	273
`MemAllocatePhysicallyContiguous`	`DriverServices.h`	376
`MemDeallocatePhysicallyContiguous`	`DriverServices.h`	378
`NanosecondsToAbsolute`	`DriverServices.h`	421
`NanosecondsToDuration`	`DriverServices.h`	421
`OpenInstalledDriver`	`Devices.h`	263
`PageInformation`	`Kernel.h`	369
`PBDequeue`	`DriverServices.h`	429
`PBDequeueFirst`	`DriverServices.h`	429
`PBDequeueLast`	`DriverServices.h`	429
`PBEnqueue`	`DriverServices.h`	429
`PBEnqueueLast`	`DriverServices.h`	429
`PBQueueCreate`	`DriverServices.h`	429
`PBQueueDelete`	`DriverServices.h`	429
`PBQueueInit`	`DriverServices.h`	429
`PoolAllocateResident`	`DriverServices.h`	375
`PoolDeallocate`	`DriverServices.h`	377
`PrepareMemoryForIO`	`Kernel.h`	360
`PStrCat`	`DriverServices.h`	431
`PStrCmp`	`DriverServices.h`	432
`PStrCopy`	`DriverServices.h`	430
`PStrLen`	`DriverServices.h`	433

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
`PStrNCat`	`DriverServices.h`	431
`PStrNCmp`	`DriverServices.h`	432
`PStrNCopy`	`DriverServices.h`	431
`PStrToCStr`	`DriverServices.h`	433
`QueueSecondaryInterruptHandler`	`Kernel.h`	411
`RegEntryID`	`NameRegistry.h`	291
`RegEntryIter`	`NameRegistry.h`	297
`RegistryCStrEntryCreate`	`NameRegistry.h`	295
`RegistryCStrEntryLookup`	`NameRegistry.h`	305
`RegistryCStrEntryToName`	`NameRegistry.h`	309
`RegistryCStrEntryToPath`	`NameRegistry.h`	308
`RegistryEntryDelete`	`NameRegistry.h`	297
`RegistryEntryGetMod`	`NameRegistry.h`	330
`RegistryEntryIDCompare`	`NameRegistry.h`	292
`RegistryEntryIDCopy`	`NameRegistry.h`	293
`RegistryEntryIDDispose`	`NameRegistry.h`	294
`RegistryEntryIDInit`	`NameRegistry.h`	292
`RegistryEntryIterate`	`NameRegistry.h`	299
`RegistryEntryIterateCreate`	`NameRegistry.h`	298
`RegistryEntryIterateDispose`	`NameRegistry.h`	304
`RegistryEntryIterateSet`	`NameRegistry.h`	298
`RegistryEntryMod`	`NameRegistry.h`	327
`RegistryEntryPropertyMod`	`NameRegistry.h`	328
`RegistryEntrySearch`	`NameRegistry.h`	301
`RegistryEntrySetMod`	`NameRegistry.h`	331

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
`RegistryEntryToPathSize`	`NameRegistry.h`	307
`RegistryPropertyCreate`	`NameRegistry.h`	311
`RegistryPropertyDelete`	`NameRegistry.h`	313
`RegistryPropertyGet`	`NameRegistry.h`	319
`RegistryPropertyGetMod`	`NameRegistry.h`	332
`RegistryPropertyGetSize`	`NameRegistry.h`	318
`RegistryPropertyIterate`	`NameRegistry.h`	315
`RegistryPropertyIterateCreate`	`NameRegistry.h`	314
`RegistryPropertyIterateDispose`	`NameRegistry.h`	317
`RegistryPropertySet`	`NameRegistry.h`	321
`RegistryPropertySetMod`	`NameRegistry.h`	333
`RegPropertyIter`	`NameRegistry.h`	314
`RemoveDriver`	`Devices.h`	270
`RenameDriver`	`Devices.h`	176
`ReplaceDriverWithFragment`	`Devices.h`	175
`ScanDriverCandidates`	`Devices.h`	254
`SendSoftwareInterrupt`	`Kernel.h`	408
`SetDriverClosureMemory`	`Devices.h`	258
`SetInterruptTimer`	`Kernel.h`	423
`SetProcessorCacheMode`	`Kernel.h`	372
`SubAbsoluteFromAbsolute`	`DriverServices.h`	421
`SubDurationFromAbsolute`	`DriverServices.h`	421
`SubNanosecondsFromAbsolute`	`DriverServices.h`	421
`SynchronizeIO`	`DriverServices.h`	373
`SysDebug`	`Kernel.h`	434

Table C-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
`SysDebugStr`	`Kernel.h`	434
`TestAndClear`	`DriverServices.h`	428
`TestAndSet`	`DriverServices.h`	428
`UpTime`	`DriverServices.h`	419
`VDDisplayConnectInfoRec`	`Video.h`	508
`VDSyncInfoRec`	`Video.h`	496
`VerifyFragmentAsDriver`	`Devices.h`	260
`VSLDisposeInterruptService`	`Video.h`	515
`VSLDoInterruptService`	`Video.h`	515
`VSLNewInterruptService`	`Video.h`	514
`VSLPrepareCursorForHardwareCursor`	`Video.h`	516

Abbreviations

Abbreviations for units of measure used in this book include

A	amperes	MHz	megahertz
cm	centimeters	mm	millimeters
dB	decibels	ms	milliseconds
GB	gigabytes	mV	millivolts
Hz	Hertz	ns	nanoseconds
KB	kilobytes	pF	picofarads
Kbit	kilobits	sec.	seconds
kHz	kilohertz	V	volts
kΩ	kilohms	W	watts
mA	milliamperes	µF	microfarads
MB	megabytes	µs	microseconds
Mbit	megabits	Ω	ohms

Other abbreviations used in this book include

ADC	analog-to-digital converter
ANSI	American National Standards Institute
AOCE	Apple Open Collaborative Environment
API	application programming interface
ASCII	American Standard Code for Information Interchange
ASIC	application-specific integrated circuit
ASLM	Apple Shared Library Manager
AV	audio/video
BIOS	basic I/O system
CD-ROM	compact disc ROM
CFM	Code Fragment Manager
CLUT	color lookup table

CPU	central processing unit
DAC	digital-to-analog converter
DAV	digital audio/video
DCE	device control entry
DDC	Display Data Channel
DEVSEL	device select
DLL	Driver Loader Library
DLPI	Data Link Provider Interface
DLSAP	data link service access point
DMA	direct memory access
DPMS	Device Power Management Standard
DSAP	destination service access point
DSL	Driver Services Library
FDDI	Fiber Distributed Data Interface
FIFO	first in, first out
FPI	family programming interface
FTP	file transfer protocol
HFS	hierarchical file system
IC	integrated circuit
ID	identifier
IDE	Integrated Drive Electronics
IDR	interrupt disabler routine
IEEE	Institute of Electrical and Electronics Engineers
IER	interrupt enabler routine
IIC	inter-IC control (also called I^2C)
I/O	input/output
IOPB	I/O parameter block
IPX	Internet Packet Exchange
ISA	Instrument Society of America
ISR	interrupt service routine
IST	interrupt source tree

LIFO	last in, first out
LSB	least significant byte
LUN	logical unit number
MPEG	Motion Picture Expert Group
MPW	Macintosh Programmer's Workshop
MSB	most significant byte
n.a.	not applicable
NC	no connection
NTSC	National Television System Committee
NVRAM	nonvolatile RAM
PAL	Phased Alternate Lines
PCI	Peripheral Component Interconnect
PCMCIA	Personal Computer Memory Card International Association
PEF	Preferred Execution Format
PLL	phase-locked loop
PRAM	parameter RAM
RAM	random-access memory
RGB	red-green-blue
RISC	reduced instruction set computing
ROM	read-only memory
SAP	service access point
SCSI	Small Computer System Interface
SECAM	Système Electronique Couleur avec Mémoire
SGR	Select Graphic Rendition
SIG	special interest group
SIM	SCSI Interface Module
SNAP	subnet access protocol
SNR	signal-to-noise ratio
SPI	system programming interface
SSAP	source service access point
TCP/IP	Transmission Control Protocol/Internet Protocol

TPI	Transport Provider Interface
VBL	vertical blanking
VCR	videocassette recorder
VESA	Video Electronics Standards Association
VGA	video graphics adapter
VIA	versatile interface adapter
VRAM	video RAM
VSL	Video Services Library
XID	exchange identifier
XPT	transport

Glossary

address invariance A feature of a data bridge (such as a **PCI bridge**) by which the address of any byte transferred across the bridge remains the same on both sides of the bridge.

address-invariant byte swapping A technique for changing data between **big-endian** and **little-endian** formats that preserves **address invariance.**

address space The domain of addresses in memory that can be directly referenced by the processor at any given moment.

address swizzling A technique for producing **address invariance** in **mixed-endian** systems by making small changes in the addresses of multibyte fields without altering the field formats—that is, without **byte swapping.**

APDA Apple's worldwide direct distribution channel for Apple and third-party development tools and documentation products.

aperture A logical view of the data in a frame buffer, organized in a specific way and mapped to a separate area of memory. For example, a frame buffer may have a **big-endian** aperture and a **little-endian** aperture, providing instant access to the buffer in either addressing mode.

Apple AV technologies A set of advanced I/O features for Macintosh computers that includes versatile telecommunications, video I/O, and 16-bit stereo sound I/O.

Apple GeoPort interface A serial I/O interface through which Macintosh computers can communicate with a variety of ISDN and other telephone transmission facilities by using external pods.

application programming interface (API) A set of services in Mac OS that supports application software. See **system programming interface.**

autoconfiguration A method of integrating peripheral devices into a computer system that includes mechanisms for configuring devices during system startup and requires that vendors include **expansion ROMs** on plug-in cards.

AV technologies See **Apple AV technologies.**

big-endian Used to describe data formatting in which each field is addressed by referring to its most significant byte. See also **little-endian.**

BIOS The Basic Input/Output System in ROM that used by Intel-compatible computers to configure devices at system startup time.

boot driver An FCode device driver that is used during the **Open Firmware startup process.** It is loaded from the **expansion ROM** on a PCI card.

bootROM A small read-only-memory (ROM) device that contains the hardware-specific code needed to start up the computer. **Open Firmware** is in the bootROM.

bridge See **PCI bridge.**

byte lane An 8-bit channel of a data bridge that passes individual bytes of data.

byte swapping A technique of changing the order of **byte lanes** as they pass through a data bridge (such as a **PCI bridge**) that produces **address invariance** in a **mixed-endian** system.

CFM See **Code Fragment Manager.**

Code Fragment Manager (CFM) A part of **Mac OS** that loads pieces of code into RAM and prepares them for execution.

coherency See **memory coherency.**

color depth The number of bits required to encode the color of each pixel in a display.

completion routine A routine provided by a Device Manager client that lets the Device Manager notify the client that an I/O process has finished.

concurrent drivers Drivers that can process more than one request at a time.

configuration The process of modifying the software of a computer so it can communicate with various hardware components.

cookie A parameter in programming that is used only to transfer a value from one routine to another.

Data Link Provider Interface (DLPI) The standard interface Apple uses for **Open Transport** drivers.

device environment A software environment with which a device operates, such as the **Open Firmware startup process** or an operating system.

Device Manager Part of **Mac OS** that installs device drivers and communicates with them.

device node In a **device tree,** a node that serves one hardware device.

device tree A software structure, generated during the **Open Firmware startup process,** that assigns nodes to all devices available to the system. **Mac OS** extracts information from the device tree to construct the device parts of the Macintosh **Name Registry.**

direct memory access (DMA) A means of transferring data rapidly into or out of RAM without passing it through the microprocessor.

disk-based driver A driver typically located in the Mac OS Extensions folder.

digital audio/video (DAV) interface A connector in certain Power Macintosh models that lets expansion cards communicate directly with the system's audio and video signal streams.

Display Manager A part of **Mac OS** that provides a uniform **family programming interface** for display devices.

DLPI See **Data Link Provider Interface.**

driver The code that controls a physical device such as a PCI card device.

driver closure A driver and all its associated libraries, for which memory may be held or released.

driver gestalt call A status call to a device driver that returns information such as the driver's revision level or the device's power consumption.

Driver Loader Library (DLL) A CFM shared library extension to the **Device Manager,** which installs and removes drivers.

Driver Services Library (DSL) A CFM shared library that supplies most of the programming interfaces required by **native drivers.** The **family programming interfaces (FPI)** provide the additional expert functions necessary for a family of devices. For example, FireWire devices belong to the FireWire family of devices, and have a FireWire FPI.

dynamic random-access memory (DRAM) Random-access memory in which each storage address must be periodically accessed ("refreshed") to maintain its value.

Expansion Bus Manager The part of the Mac OS startup software that provides access to I/O memory and manages the storage of certain information in **nonvolatile RAM.**

expansion ROM A ROM on a PCI expansion card that supplies the computer with information about the card and any associated peripheral devices during the configuration process. Also called a *declaration ROM* or a *configuration ROM.* The expansion ROM can contain **FCode** which is used by **Open Firmware** during the Macintosh startup process.

expert The code that connects a family of devices to the **native I/O framework.**

family A collection of devices that provide the same kind of functionality, such as the set of **Open Transport** devices.

family administrator Code that sends configuration information to a family of devices.

family expert An expert that uses the **Name Registry** to find device entries of its family service type.

family library A set of routines that a **family expert** uses to support PCI devices of its family service type.

family programming interface (FPI) A set of system services that mediate between **family experts** and the devices within a family.

Fast Path An optional optimization of **Open Transport** wherein the driver supplies the client with a precomputed packet header for a given destination.

FCode A tokenized version of the Forth programming language. For example, FCode is used in PCI card **expansion ROMs.** The elements of FCode are all 1 or 2 bytes long.

FCode tokenizer A utility program that translates lines of Forth source code into **FCode.**

frame buffer Memory that stores one or more frames of video information for display on a screen.

gestalt node A node at the root of the **device tree** that contains information about the Macintosh system.

GeoPort See **Apple GeoPort interface.**

hard decoding The practice by which an expansion card defines PCI **address spaces,** instead of letting the Macintosh system assign relocatable base addresses.

hardware interrupt A physical device's method for requesting attention from a computer.

hardware interrupt handler The part of an interrupt handler that responds directly to a hardware interrupt. It usually satisfies the source of the interrupt and queues a **secondary interrupt handler** to perform the bulk of the interrupt servicing.

hardware interrupt level The execution context provided to a device driver's **hardware interrupt handler.** In this context hardware interrupts of the same or lower priority are disabled.

IEEE Institute of Electrical and Electronics Engineers.

input/output (I/O) Parts of a computer system that transfer data to or from peripheral devices.

installation Of an interrupt, the process of associating an **interrupt source** with an **interrupt handler.**

interrupt dispatching The process of invoking an **interrupt handler** in response to an interrupt.

interrupt handler Code that performs tasks required by a **hardware interrupt.**

interrupt registration The process of attaching an interrupt handler to the **interrupt source tree.**

interrupt set One level in an interrupt tree.

interrupt source A physical device that is able to interrupt the process flow of the computer.

interrupt source tree (IST) A data structure associated with a hardware **interrupt source** that contains the interrupt handling routines that the Macintosh system may execute.

little-endian Used to describe data formatting in which each field is addressed by referring to its least significant byte. See also **big-endian.**

low-level expert An expert that places information about devices in the **Name Registry.**

Macintosh Programmer's Workshop (MPW) A complete software development environment that runs on Macintosh computers.

Mac OS Apple's operating system software for Macintosh and Macintosh-compatible computers. Previously called *Macintosh system software.*

methods In the context of this document, methods are parts of an Open Firmware device package that perform operations like a function, subroutine, or procedure.

memory coherency The property of a range or kind of memory by which all parts of the computing system access the same values. Memory coherency ensures that data being moved into or out of memory does not appear to have different values when accessed by the processor and **PCI bridges.**

mixed-endian The ability of a computer system, such as Power Macintosh, to support both **big-endian** and **little-endian** data formats.

modifier Information associated with a name or **property** that is hardware or implementation specific, such as whether or not the name or property is saved to **nonvolatile RAM.**

name entry An element of the **Name Registry**. Name entries are connected hierarchically to other name entries and have **properties.**

Name Registry A high-level **Mac OS** system service that stores the names of software objects and the relations among the names. The Name Registry extracts device information from the **device tree** and makes it available to Macintosh **run-time drivers.**

native driver A driver that is written in PowerPC code and that uses the **native I/O framework** in PCI-based Power Macintosh computers.

native driver package A **CFM** code fragment that contains the driver software for a family of devices.

native I/O framework The set of services in **Mac OS** that support native run-time drivers.

noninterrupt level See **task level.**

nonvolatile RAM (NVRAM) Memory, in either flash ROM or battery-powered RAM, that retains data between system startups.

Open Firmware driver An FCode driver utilized to support devices during the Macintosh startup process. See **boot driver.**

Open Firmware startup process The startup process by which PCI-compatible Macintosh computers recognize and configure peripheral devices connected to the **PCI local bus.** It conforms to an IEEE standard 1275.

Open Transport A device family that handles network devices such as LocalTalk and Ethernet.

pass-through memory cycle A PCI data transfer cycle in which the **PCI bridge** passes the original PowerPC word address to the **PCI bus.**

PCI Abbreviation for *Peripheral Component Interconnect.*

PCI bridge An ASIC chip that communicates between the computer's microprocessor and a **PCI local bus.**

PCI local bus A bus architecture for connecting ASICs and plug-in PCI expansion cards to a computer's main processor and memory. It is defined by the **PCI specification.**

PCI specification *PCI Local Bus Specification,* Revision 2.0, a document issued and maintained by the PCI Special Interest Group.

physical device A piece of computer hardware that performs an I/O function and is controlled by a driver.

pixel A single dot on a screen display.

port driver A driver for **Open Transport.**

PowerPC A family of RISC microprocessors.

property A piece of descriptive information associated with a node in the device tree or with a name entry in the **Name Registry.**

property list The collection of **properties** associated with a device.

reduced instruction set computing (RISC) A technology of microprocessor design in which all machine instructions are uniformly formatted and are processed through the same steps.

RISC See **reduced instruction set computing.**

ROM-based driver A driver located in the **expansion ROM** of a PCI expansion card.

run-time driver A device driver that is used by an operating system after the **Open Firmware startup process** has finished. It may be supplied by the operating system, contained in an **expansion ROM** on a PCI expansion card, or a disk-based driver.

scanning The process of matching a device with its corresponding driver.

scatter-gather buffer A buffer that stores data in several discontiguous ranges of memory.

scatter-gather list The set of physical address ranges corresponding to a transfer buffer.

SCSI Interface Module (SIM) A driver for a SCSI-bus host adapter that is compatible with SCSI Manager 4.3. There are also SCSI drivers and SCSI disk drivers.

secondary interrupt handler An **interrupt handler** that is queued for execution after the **hardware interrupt handler** has responded to the interrupt. Secondary interrupt handlers can be interrupted and execute serially when the system is not otherwise busy.

secondary interrupt level The execution context provided to a device driver's **secondary interrupt handler.** In this context hardware interrupts are enabled and additional interrupts may occur.

transversal interrupt service routine (ISR) routes interrupt processing from a member to one of its child members. Transversal ISRs are always attached to root or parent/child members.

SIM See **SCSI Interface Module.**

SPI See **system programming interface.**

startup firmware Code in the Macintosh ROM that implements the **Open Firmware startup process.** The NewWorld architecture implements a **bootROM** that contains Open Firmware FCode that performs the startup process.

task level The execution environment for applications and other programs that do not service interrupts. Also called *noninterrupt level.*

time base The model-dependent rate on which real-time timing operations are based in the **Driver Services Library (DSL)**.

vertical blanking task A task that executes during a display device's vertical retrace intervals.

virtual device I/O code that provides a capability that is not hardware specific—for example, a RAM disk.

YUV A format for representing video data.

Index

T H E   A P P L E   P U B L I S H I N G   S Y S T E M

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

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

WRITERS
George Towner, Steve Schwander

DEVELOPMENTAL EDITORS
Jeanne Woodward, Beverly Zegarski, Laurel Rezeau

ILLUSTRATORS
Sandee Karr, David Arrigoni

PRODUCTION EDITORS
Lorraine Findlay, Gerri Gray

Special thanks to Jano Banks, Clinton Bauder, Mark Baumwell, Öner Biçakçi, Bill Bruffey, Mike Cappella, Derrick Carty, Mark Cookson, David Ferguson, John Fitzgerald, Kit Fitzpatrick, Wayne Flansburg, Ian Hendry, Ron Hochsprung, Fred Janon, Holly Knight, Ben Koning, Al Kossow, Rich Kubota, Tom Mason, Martin Minow, John Mitchell, Matthew Nelson, Jon Norenberg, Gary Phillips, Steve Polzin, Mark Pontarelli, Mike Puckett, Mike Quinn, Dave Radcliffe, Paul Resch, Tom Saulpaugh, Erik Staats, Carl Sutton, Larry Thompson, Fernando Urbina, Allen Watson, and Tony Wingo.