Interim Draft

Designing PCI Cards and Drivers for Power Macintosh Computers

February 10, 1995
Apple Developer Press

A7 DRAFT—PRELIMINARY INFORMATION

This draft contains preliminary information about forthcoming Apple technology. It is intended to provide advance guidance to designers of equipment compatible with Power Macintosh computers. The information in this draft is subject to change, and no representation or guarantee is made about its accuracy or completeness.

Apple Computer, Inc.

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

Simultaneously published in the United States and Canada.

About This Draft

This draft of *Designing PCI Cards and Drivers for Power Macintosh Computers* is an interim snapshot of a work in progress. It has been compiled specifically to accompany a preliminary version of the Macintosh PCI Developer's Kit.

Many of the system services described in this draft are still under development. Additional services are not yet available for release. For the latest and most accurate information, contact Apple Computer directly.

Differences from Previous Drafts

The last draft of this document generally distributed to developers was dated October 24, 1994. The current draft incorporates the DAV documentation, previously distributed separately, as Part 4.

The paper version of this draft contains change bars that mark the principal differences in content between this draft and the October 24 draft. The online version does not contain change bars.

Contents

Figures, Tables, and Listings

About This Book

This book describes the Macintosh implementation of the Peripheral Component Interconnect (PCI) local bus established by the PCI Special Interest Group. 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 product developers who want to design PCI expansion cards and their associated software to be compatible with the newer computers. It also includes technical details of the digital audio/video (DAV) interface that some Power Macintosh models provide for PCI cards.

The information in this book is general. You should also refer to the developer notes that accompany each Macintosh product release for exact details of that product's PCI and DAV implementations.

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" beginning on page xxiii.

Contents of This Book

This book is divided into four parts and contains 18 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, "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.
- Chapter 3, "Data Transfers," describes the processes of data movement over the PCI bus.

System Startup By Open Firmware

Part 2, "The Open Firmware Process," describes the startup process in Power Macintosh computers that support the PCI bus and run Mac OS:

- 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 3, "Native PCI Card Drivers," tells you how to design and write runtime PCI card drivers for the second generation of 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, "Writing Native Drivers," gives you details of native driver design and coding.
- Chapter 8, "Macintosh Name Registry," describes the Mac OS data structure that stores device information extracted from the PCI device tree.
- Chapter 9, "Driver Services Library," details the general services that Mac OS provides for device drivers, including interrupt services.
- Chapter 10, "Expansion Bus Manager," discusses a collection of timing and other bus-specific system services available to native device drivers.
- Chapter 11, "Graphics Drivers," describes the calls serviced by typical display drivers.
- Chapter 12, "Network Drivers," describes the construction of a sample network driver.
- Chapter 13, "SCSI Drivers," describes the construction of a sample native SCSI Interface Module (SIM) compatible with Macintosh SCSI Manager 4.3.

Accessing Audio and Video Data

Part 4, "The DAV Interface," describes the hardware and software characteristics of the Macintosh digital audio/video interface in certain models of PCI-based Power Macintosh computers. The DAV feature lets expansion cards access the computer's digital sound and video data streams independently of the PCI bus. Part 4 consists of the following chapters:

- Chapter 14, "Accessing AV Data," describes the general mechanical and electrical characteristics of the DAV interface and summarizes the system software that Apple supplies for DAV signal management.
- Chapter 15, "Audio Hardware Interface," provides electrical and timing details of the digital sound I/O signals that are present at the DAV interface.
- Chapter 16, "Audio Programming," describes the Macintosh system programming interface (SPI) that lets expansion cards, drivers, and applications control the sound signals at the DAV interface.
- Chapter 17, "Video Hardware Interface," provides electrical and timing details of the video I/O signals at the DAV interface.
- Chapter 18, "Video Programming," describes the Macintosh SPI that lets expansion cards, drivers, and applications control the video signals at the DAV interface.

Help For Developers

An appendix, "Development Tools," follows the main part of this book. It describes the developer's kit that Apple supplies for designing PCI cards and related software compatible with Power Macintosh computers. It also lists all the routines and data structures documented 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 Developer Press publishes a variety of books and technical notes designed to help third-party developers design hardware and software products compatible with Apple computers.

Inside Macintosh is a collection of books, organized by topic, that describe the system software of Macintosh computers. Together, these books provide the essential reference for programmers, software designers, and engineers. They include the following titles:

Inside Macintosh: AOCE Application Interfaces
Inside Macintosh: AOCE Service Access Modules
Inside Macintosh: Devices
Inside Macintosh: Files
Inside Macintosh: Imaging With QuickDraw
Inside Macintosh: Interapplication Communication

Inside Macintosh: Macintosh Toolbox Essentials
Inside Macintosh: Memory
Inside Macintosh: More Macintosh Toolbox
Inside Macintosh: Networking
Inside Macintosh: Operating System Utilities
Inside Macintosh: Overview
Inside Macintosh: PowerPC Numerics
Inside Macintosh: PowerPC System Software
Inside Macintosh: Processes
Inside Macintosh: QuickDraw Environment and Utilities
Inside Macintosh: QuickDraw Graphics
Inside Macintosh: QuickDraw Objects
Inside Macintosh: QuickDraw Printing
Inside Macintosh: QuickDraw Printing Extensions and Drivers
Inside Macintosh: QuickDraw Utilities
Inside Macintosh: QuickTime
Inside Macintosh: QuickTime Components
Inside Macintosh: Sound
Inside Macintosh: Text

Inside Macintosh: Devices documents the last version of the Device Manager before its enhancements to support PowerPC native drivers. It also contains a full description of SCSI Manager 4.3.

Inside Macintosh: PowerPC System Software covers in detail the changes and extensions to Macintosh system software version 7.1 for Power Macintosh computers, including new Macintosh Toolbox managers and the run-time architecture that supports the PowerPC microprocessor.

Building Programs for Macintosh With PowerPC is a general discussion for developers of the development and building of application software for PowerPC processor–based Macintosh systems, including Power Macintosh computers that use the PCI bus.

Designing Cards and Drivers for the Macintosh Family, third edition, explains the general software requirements for drivers compatible with Macintosh computers.

Technical Introduction to the Macintosh Family, second edition, surveys the complete Macintosh family of computers from the developer's point of view.

Macintosh Human Interface Guidelines provides authoritative information on the theory behind the Macintosh "look and feel" and Apple's standard ways of using individual interface components. A companion CD-ROM disk, *Making It Macintosh,* illustrates the Macintosh human interface guidelines through interactive, animated examples.

Macintosh Developer Note Number 5 gives technical details of the Macintosh Quadra 660AV and 840AV computers, which introduced the advanced audio and video capabilities called *Apple AV technologies.*

Macintosh Developer Note Number 8 contains two documents: *Power Macintosh Computers* describes the Power Macintosh 6100/60, 7100/66, and 8100/80 models; *Macintosh DAV Interface for NuBus Expansion Cards* contains hardware details of the DAV interface provided for NuBus™-based Macintosh computers, including the Macintosh Quadra 660AV and 840AV and the Power Macintosh 7100/66AV and 8100/80AV.

Display Device Driver Guide describes device support for the Macintosh Display Manager. It was published in electronic form on the December, 1994, Developer CD.

Macintosh New Technical Notes HW-30 describes Apple's revisions to the way that Macintosh computers automatically sense video display characteristics.

Technical Note 144 (*Macintosh Color Monitor Connections*), Technical Note 326 (*M.HW.SenseLines*), and *Macintosh New Technical Note HW-30* provide technical details of the interfaces to various Apple and third-party monitors.

Most of the Apple publications just listed are available from APDA. APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.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

Comité Consultatif International Radio (CCIR):

The CCIR standards organization publishes *Recommended Standard 601-2,* a full technical description of the YUV digital video format used in the Macintosh DAV interface.

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

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.)
908-562-5432 (International)

IT&T:

The IT&T *ASCO 2300 Audio-Stereo Codec Specification* defines the operation of the Macintosh AWAC sound codec connected to the DAV interface. It is available direct from IT&T.

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.)
503-797-4207 (International)
Fax 503-234-6762

The PCI SIG also publishes *PCI Multimedia Design Guide*.

Philips:

Philips publishes several manuals that describe the chips used to process video signals in Power Macintosh computers:

Desktop Video Data Handbook (1994)
SAA7196 Digital Video Decoder, Scaler, and Clock Generator
TDA8758 YC 8-Bit Low-Power Analog-to-Digital Video Interface

These manuals are available direct from Philips.

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 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 that begins on page 353.

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

Hexadecimal numbers are preceded by a dollar sign ($). For example, the hexadecimal equivalent of decimal 16 is written as $10.

Note
A 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

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	µF	microfarads
mA	milliamperes	µs	microseconds
MB	megabytes	Ω	ohms
Mbit	megabits		

Other abbreviations used in this book include

ADC	analog-to-digital converter
AGND	analog ground
ANSI	American National Standards Institute
API	application programming interface
ASCII	American Standard Code for Information Interchange
ASIC	application-specific integrated circuit
ASLM	Apple Shared Library Manager
AV	audio/video
AWAC	audio waveform amplifier and converter
BIOS	basic I/O system
CCIR	Comité Consultatif International Radio
CD-ROM	compact disc ROM
CFM	Code Fragment Manager
CLUT	color lookup table
CMOS	complementary metal-oxide silicon
CPU	central processing unit
DAC	digital-to-analog converter
DAV	digital audio/video

DCE	device control entry
DDC	Display Data Channel
DEVSEL	device select
DLSAP	data link service access point
DLL	Driver Loader Library
DLPI	Data Link Provider Interface
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
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
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 Programmers 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
rms	root mean square
ROM	read-only memory
SCSI	Small Computer System Interface
SECAM	Système Electronique Couleur avec Mémoire
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
TBD	to be determined
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
XID	exchange identifier
XPT	transport

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, "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.
- Chapter 3, "Data Transfers," describes the processes of data movement over the PCI bus.

Later parts of this book cover the following topics:

- Part 2, "The Open Firmware Process," describes the startup process in Power Macintosh computers that support the PCI bus and run Mac OS. Part 2 begins on page 27.
- Part 3, "Native PCI Card Drivers," tells you how to design and write runtime PCI card drivers for the second generation of Power Macintosh computers. Part 3 begins on page 43.
- Part 4, "The DAV Interface," describes the hardware and software characteristics of the Macintosh digital audio/video interface in certain models of PCI-based Power Macintosh computers. Part 4 begins on page 301.

P A R T O N E

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. The second generation of Power Macintosh computers, containing PowerPC microprocessors, uses PCI 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 2.0, 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 *PCI Local Bus Specification,* Rev. 2.0, 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 33 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

The PCI bus exhibits a number of fundamental differences with 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
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
Timeout	255 bus clocks	5 bus clocks
Burst capability	8, 16, 32, or 64 bytes	Any number of bytes
Power allocation	15 watts per card	7.5, 15, or 25 watts per card

The Macintosh Implementation of PCI

To achieve maximum compatibility with PCI-compliant devices and plug-in cards, the second generation of Power Macintosh computers is designed to comply with the *PCI Local Bus Specification,* Revision 2.0. 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

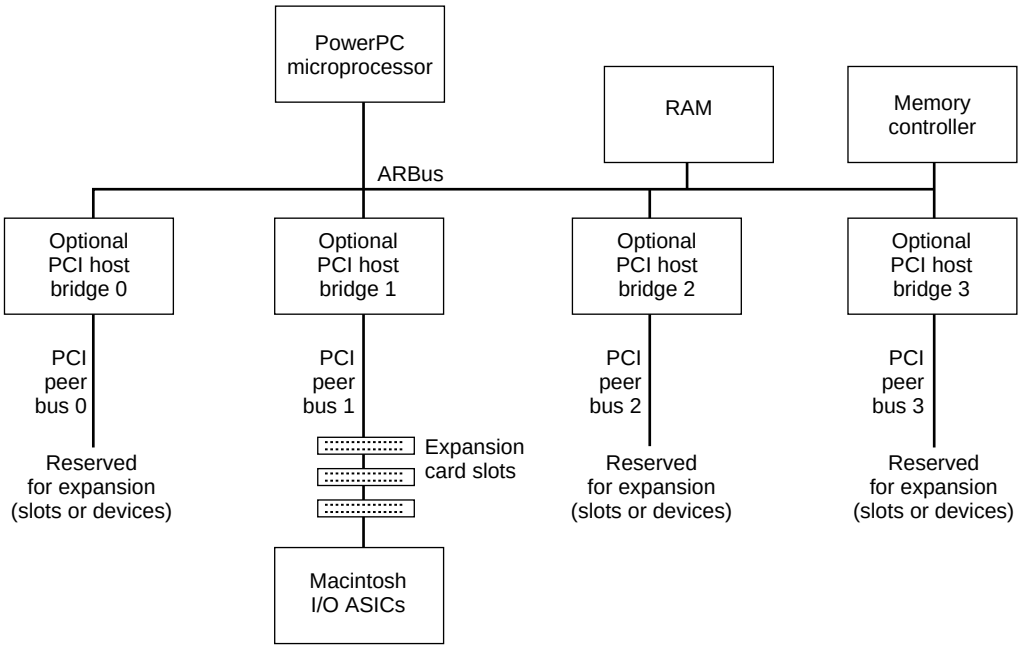
As explained in "Address Allocations" on page 16, a Power Macintosh computer may contain as many as four separate PCI buses for expansion cards, although initial models contain fewer than four.

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 first 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



PCI Bus Characteristics

The PCI bus on Power Macintosh follows the requirements of the PCI specification described on page xxvi. 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)
Address/data bus width	32 bits
Signal voltage	5 V
PCI address spaces supported	Memory, IO, and configuration
Minimum power supplied	5 V rail: 3 A (15 W) per slot[1] 3.3 V rail: 2 A (6.6 W) per slot[1]
PCI bus arbitration	Fair, round-robin, all slots master-capable
Mechanical bracket	ISA style
Plug-in card expansion ROM	Highly recommended[2]
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[3]
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 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.

[2] 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 works) can be provided only when an expansion ROM contains both startup firmware and run-time driver code. See Chapter 4, “Startup and System Configuration,” for more information on expansion ROM benefits, contents, and data formats.

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

Semaphores must be maintained in main system memory through processor control, using the PowerPC `lwarx` and `stwcx` instructions. C programs can access semaphores

by using the routines described in "Atomic Memory Operations" beginning on page 213. Power Macintosh does not support the use of semaphores in PCI memory space.

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.
- 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" beginning on page 17.
- 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

Table 1-3 Bridge support for PCI cycle types (continued)

Lines C/BE#[3:0]	Command	Supported as PCI master	Supported as PCI target
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 Non-Memory Space Cycle Generation" beginning on page 224. 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 I/O space. The first implementation of the PCI bus for PowerMacintosh 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 space. 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 I/O cycles must go through a system programming interface, and 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 system programming interface for I/O cycles is described in "I/O Space Cycle Generation" beginning on page 224.

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 system programming interface, high performance interfaces should try to use the PCI memory space instead of configuration space. The system programming interface for configuration cycles is described in "Configuration Space Cycle Generation" beginning on page 228.

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 system 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 system programming interface for interrupt acknowledge cycles is described in "Interrupt Acknowledge Cycle Generation" beginning on page 232.

Special Cycles

Special cycles are generated by using a system programming interface that causes a special cycle (write) on the PCI bus. The special cycle transmits the data message passed to the interface. The system programming interface for special cycles is described in "Special Cycle Generation" beginning on page 233.

Maximizing Bus Performance

The guidelines in this section can help you 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

Note
The current PowerPC architecture has a cache line size of 32 bytes. ◆

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 1-4.

Table 1-4 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 1-5.

Table 1-5 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 signalled, because the write was already posted).
Read	Bus Error	Generate Target Abort. Signaled Target Abort Error Interrupt generated.

Expansion Card Characteristics

Every PCI expansion card should contain code in its expansion ROM conforming to IEEE Standard 1275. The requirements for this code (and the benefits of its inclusion in expansion ROMs) are discussed in "The Open Firmware Startup Process" beginning on page 30.

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" on page 20.

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 Manager task may be installed. For more information on this subject, see Chapter 11, "Graphics Drivers."

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

▲ **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. ▲

Hard Decoding

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 been allowed to hard decode (for example, VGA and IDE applications). Hard decoding cripples the ability of system software to resolve address conflicts between devices. A problem exists when multiple devices which hard decode the same 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.

In a Macintosh system it is very common, for example, 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 recommended, 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.

If hard decoding cannot be turned off in the device, a fixed address `reg` property entry must be entered into the device tree by the device’s FCode. But the best way to ensure that your card will always be allocated address space is to avoid hard decoding.

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” on page 221.

An important use of the Power Macintosh NVRAM is to store the `little-endian?` variable, discussed in “Addressing Mode Determination” on page 20.

Access to Apple AV Technologies

Certain PCI-based Power Macintosh models are equipped with a group of advanced audio and video I/O features called ***Apple AV technologies.*** These features include

- versatile access to voice, fax, and data services through the ***Apple GeoPort interface***
- video input and output capabilities compatible with both S-video and composite video in NTSC, PAL, and SECAM formats
- broadcast-quality 16-bit stereo sound input and output
- speech recognition and synthesis

Power Macintosh computers with these features include a connector, available to PCI expansion cards, that supports the Macintosh ***digital audio/video (DAV) interface.*** The DAV interface gives a PCI card direct access to the Macintosh system's unscaled YUV video input signal and audio data stream. PCI cards that use the DAV connector can exchange audio and video signals with the Macintosh system without having to pass these data through the PCI bus. YUV digital video format is described in CCIR *Recommended Standard 601-2,* listed in "Other Publications" beginning on page xxv.

The Macintosh DAV interface for PCI expansion cards, including its control software, is described in Part 4, "The DAV Interface," beginning on page 301.

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. The processes of data transfer over PCI buses are described in Chapter 3, "Data Transfers."

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" beginning on page 8.
- After each PCI host bridge, PCI-to-PCI bridges may be added in any configuration to create up to 256 PCI buses in a Power Macintosh computer, the maximum that the PCI specification allows.
- 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.

Table 2-1 Power Macintosh memory allocations

Address range	Usage
$0000 0000–$7FFF FFFF	System RAM
$8000 0000–$EFFF FFFF	Available to PCI expansion cards
$F000 0000–$F1FF FFFF	PCI host bridge 0 control
$F200 0000–$F3FF FFFF	PCI host bridge 1 control
$F400 0000–$F5FF FFFF	PCI host bridge 2 control
$F600 0000–$F7FF FFFF	PCI host bridge 3 control
$F800 0000–$F8FF FFFF	System control
$F900 0000–$FEFF FFFF	Available to PCI expansion cards
$FF00 0000–$FFFF 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 10, "Expansion Bus Manager." Using these routines, you can create all types of data transactions on Macintosh PCI buses in a hardware-independent way.

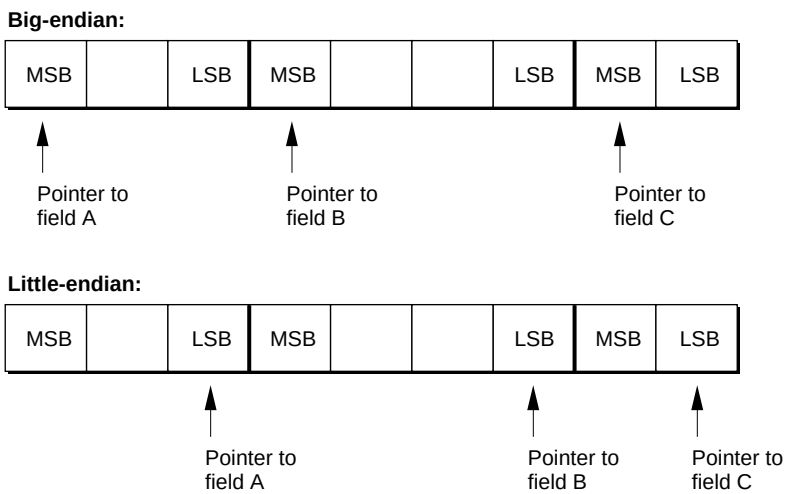
Addressing Modes

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
- ***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



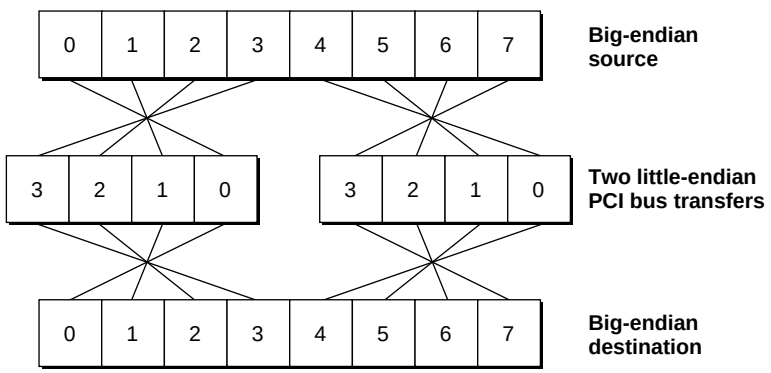
Since data fields are normally stored in RAM by writing from lower to higher addresses, big-endian addressing also means that the field's lowest address in physical memory contains its most significant byte; little-endian addressing means that the field's lowest address contains its least significant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

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 invariance 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 the routines described in "Atomic Memory Operations" beginning on page 213.
- 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.

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 is 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 following code makes this 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" on page 13.

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

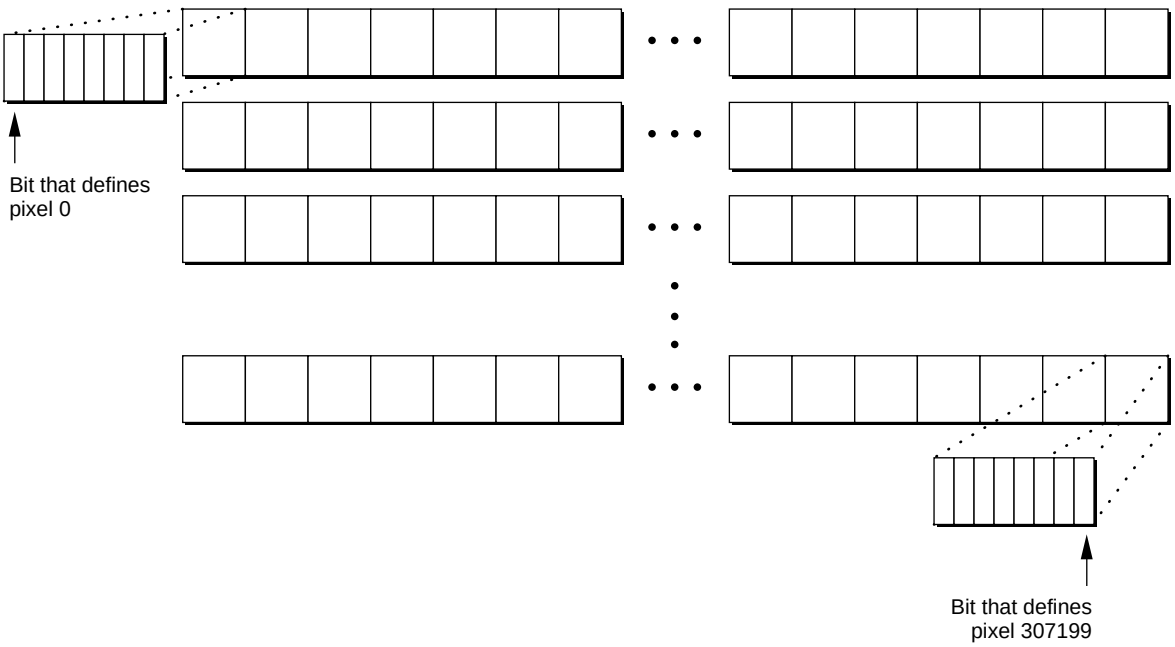
The Macintosh pixel storage format is big-endian. This format has the following general characteristics:

- 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 a description of how frame buffer data is transported over the PCI bus, see "Data Flow" on page 24. For further information about Macintosh pixel formats, see Chapter 13 in *Technical Introduction to the Macintosh Family.* This book is listed in "Supplementary Documents," in the preface.

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 arragement 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 the byte-swapping operations described in “Data Flow” on page 24. 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 xxvi.

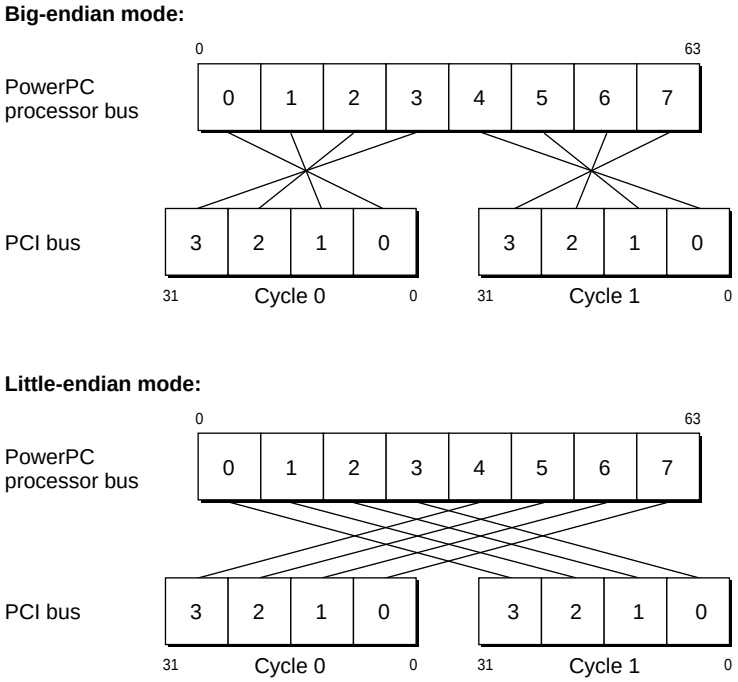
Data Transfers

Power Macintosh computers support PCI buses in conformance with the PCI specification, which is listed in "Supplementary Documents" beginning on page xxiii. This chapter explains how Power Macintosh computers accomplish the processes of data movement described in the specification, including the ways that PCI bus cycles work in the Power Macintosh environment.

Data Flow

As discussed in Chapter 2, the PowerPC processor bus in Power Macintosh computers uses big-endian addressing when running a big-endian operating system such as Mac OS. The PCI bridge chip that interconnects the PowerPC processor bus and the little-endian PCI bus performs the necessary byte swapping, using the mechanisms described in "Addressing Modes" beginning on page 17. Based on the addressing mode of the operating system, the bridge chip can be configured by system software to be run with the PowerPC set in either big-endian or little-endian mode. In either setting, the bridge correctly maintains address invariance with respect to the constantly little-endian PCI bus. The resulting data transfer patterns are shown in Figure 3-1, where the numbers in the boxes indicate byte ordering.

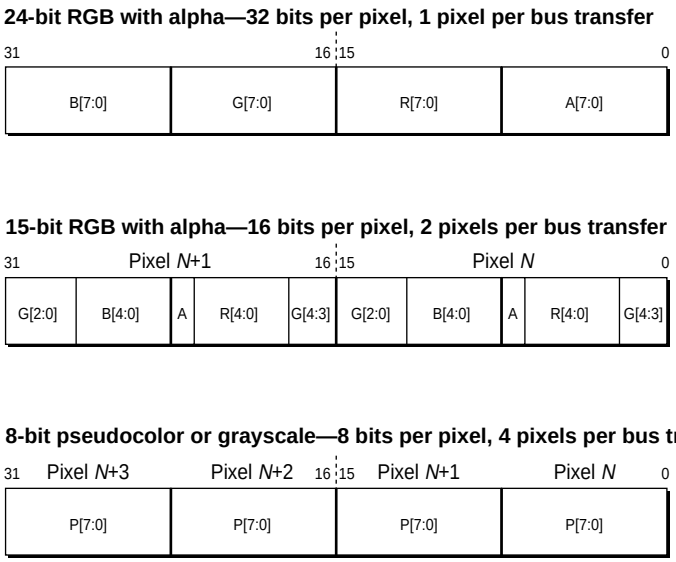
Figure 3-1 Big-endian and little-endian data transfers



Note
In Figure 3-1, little-endian processor mode is shown only for completeness; it is not used when Macintosh computers run Mac OS. ◆

When accessing video and graphics frame buffers, Mac OS assumes that they store data in the big-endian pixel format described in "Frame Buffers" on page 20. Figure 3-2 shows MacOS RGB and grayscale formats after big-endian to little-endian byte-swapping has been performed by the PCI host bridge.

Figure 3-2 Mac OS frame buffer contents byte-swapped to the PCI bus



Data Transfer Cycles

The PCI bus transfers data by means of memory, I/O, configuration, interrupt acknowledge, and special cycles, in accordance with the PCI specification. Power Macintosh computers generate PCI memory cycles for all the address spaces listed as available to PCI expansion cards in Table 2-1 on page 16. They also generate I/O, configuration, interrupt acknowledge, and special cycles through reserved memory-mapped spaces in the PCI host bridge control spaces. The Power Macintosh implementation of these cycles is discussed in more detail in the next sections.

Note
To ensure future compatibility, designers of drivers and expansion card firmware must use the calls described in Chapter 10 to create I/O, configuration, interrupt acknowledge, and special cycles. ◆

The PCI Bus and Open Firmware

Adopting the PCI bus gives Power Macintosh computers a new level of compatibility with third-party hardware devices. To provide equivalent software compatibility, Power Macintosh computers that implement the PCI bus also support the IEEE standard Open Firmware process of system startup.

During the Open Firmware process, startup firmware in the Macintosh computer's ROM searches the PCI buses and generates a data structure that lists all available peripheral devices. This data structure also stores the support software, including drivers, provided by each PCI expansion card. The startup firmware then finds an operating system 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 PCI-compatible Power Macintosh computers to operate PCI peripheral devices using either Macintosh or third-party system software.

The Open Firmware process in the second generation of Power Macintosh computers is described in the next part of this book.

PART TWO

The Open Firmware Process

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

- 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.

P A R T T W O

Startup and System Configuration

This chapter describes the Open Firmware startup process by which PCI-compatible Power Macintosh computers recognize and configure peripheral devices connected to the PCI expansion card bus. As explained in "The PCI Bus and Open Firmware" on page 26, 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 code type and the Open Firmware type. Apple has chosen the Open Firmware type because it has wide industry acceptance and will let Power Macintosh computers run nearly any operating system.

PCI cards for Power Macintosh computers must include expansion ROMs and should support the Open Firmware startup process, at least to the extent necessary for the cards to be installed at startup time. Cards that need to operate I/O devices during the Open Firmware startup process require more than the minimum level of support, as described in "Device Installation" beginning on page 32.

The Open Firmware Startup Process

The ***Open Firmware startup process*** in PCI-compatible Power Macintosh computers 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 currently available on request from AppleLink address APPLE.PCI; *IEEE Standard 1275* is described in "Supplementary Documents" beginning on page xxiii.

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 PCI cards, called ***expansion ROMs.*** While the startup firmware is running, the Power Macintosh computer starts up and configures its hardware (including peripheral devices) 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 is written in the Forth language, as defined by IEEE Standard 1275. 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 so that drivers can run on the PowerPC main processor. Expansion card firmware can modify the Open Firmware startup process by supplying FCode that the computer's startup firmware loads and runs before launching an operating system.

- The startup firmware creates a data structure of nodes called a ***device tree,*** in which each device is described by a ***property list.*** The device tree is stored in system RAM. 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 8. The full list of standard device tree properties is given in IEEE Standard 1275; the properties that Mac OS uses are listed in Table 8-1 on page 156.
- Device drivers that are required during system startup (called ***Open Firmware drivers***) 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 drive support resources such as fonts. The startup firmware in the Power Macintosh ROM installs Open Firmware drivers in system RAM and lets them run on the PowerPC main processor.
- The startup firmware in the Power Macintosh ROM contains debugging facilities for both FCode and some kinds of operating-system code. These facilities can help expansion card designers develop the firmware for new peripheral devices compatible with Macintosh computers.

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 one-to-one. The Forth vocabulary that you can use is presented in IEEE Standard 1275. For a list of some of the Apple and third-party tools available to help you write PCI card firmware in Forth, see Appendix A, "Development Tools."

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, all device drivers specific to that environment should be stored 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 7, "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

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.

Device Installation

If Open Firmware boot code is not included in a card's expansion ROM, the card is not usable during system startup and must be separately installed by the operating system after startup. The possible ways that a device with a valid PCI expansion ROM can be installed include the following:

- ***Expansion ROM contains run-time drivers and full Open Firmware support, including an Open Firmware driver.*** In this case, the expansion device can be used at startup time. The driver code for the device either can be loaded from the card's firmware. This option requires more extensive code in the card's expansion ROM, because the card must be able to initialize itself and provide driver code to the Macintosh system before any operating system is running.
- ***Expansion ROM contains run-time drivers and minimal Open Firmware support.*** If a PCI card is not needed at startup time, only a small amount of Open Firmware boot code is required to install its ROM-based device properties in the device tree and, in certain cases, perform some device initialization. Such properties might include run-time device drivers that are specific to various operating systems.

- ***Expansion ROM contains no FCode.*** 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. Driver code must be loaded from a boot 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.

Open FirmwareDriver Support

As explained in "Startup Firmware" on page 30, 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. In addition, the Macintosh startup firmware does not provide hardware interrupts; Open Firmware drivers must detect external events by polling devices. However, the startup firmware in some Power Macintosh ROMs may contain hardware-specific support packages that Open Firmware drivers can use for common tasks.

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 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 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.

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. PCI cards may simultaneously perform hardware-specific initialization and test procedures at this time.

First, the Power Macintosh startup firmware polls the computer's PCI buses, interrogating addresses where devices might be found. Each time it finds 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 from the PCI card 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. These properties always include the device name and usually also include some or all of the information specified by IEEE Standard 1275.

An important set of device tree properties include Open Firmware drivers for PCI devices; other drivers 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.

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 device
- a mass storage device containing operating-system code

The Power Macintosh startup firmware then continuously polls the input device for characters, writes output characters to the screen, and calls the load routine for the mass storage device. To do this, it uses FCode drivers copied from the expansion ROMs for these devices. When the load routine finishes successfully, the Power Macintosh startup firmware uses the Forth `go` command to start execution of an operating system—in this example, Mac OS.

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.

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.

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 Three, "Native PCI Card Drivers."

General Requirements

Open Firmware drivers, which run during the Open Firmware startup process, must be written in FCode, so that the Macintosh firmware can run them in the absence of an operating system. For further information about FCode, see *Writing FCode Programs for PCI*. This book is listed in "Other Publications," beginning on page xxv.

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.
- They should avoid addressing system space below 1 MB; this space is generally reserved for operating system and application use.
- PCI expansion cards and their drivers should avoid hard address decoding, as discussed in "Hard Decoding" on page 13.
- Cards that consume more than 3 A at 5 V or 2 A at 3.3 V should implement the high and low power options discussed in "Card Power Controls" on page 235.

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 PCI Card Drivers," beginning on page 43.

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.

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 driver written in FCode. When it finds such a driver, the Open Firmware startup process loads it into RAM and interprets and executes the FCode. 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.* This book will ultimately be available from the PCI SIG; at present you can request a copy by sending a message to AppleLink address APPLE.PCI.

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," beginning on page xxiii.

Standard device properties for PCI expansion cards and run-time drivers used with Mac OS are listed in Table 8-1 on page 156. 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.

Developing and Debugging Open Firmware Code

This section discusses certain facilities included in the firmware of the second generation of Power Macintosh computers that help developers design and debug Open Firmware code in the absence of a running operating system. For further information about Open Firmware development tools, see Appendix A, "Development Tools."

Terminal Emulation

The Open Firmware standard (described in "Supplementary Documents," beginning on page xxiii) defines the behavior of a terminal emulator support package, including the implementation of certain escape sequences from the set 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 6429-1983 standard 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, each of whose value is 0-15, 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=15 and foreground=0, thus producing black characters on a bright white background.

Table 5-1 describes the effect of executing SGR escape sequence with various parameters.

Table 5-1 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 5-1 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 5-2, where values are defined in terms of the fraction of full saturation required for each of the primary red-green-blue (RGB) colors.

Table 5-2 Color table values

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

Table 5-2 Color table values (continued)

Index	Red	Green	Blue	Color
12	1	1/3	1/3	Light Red
13	1	1/3	1	Light Magenta
14	1	1	1/3	Yellow
15	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 extenstions. 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. 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..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 the index color, 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 the index color, 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.

User Interface

The Macintosh implementation of Open Firmware includes a user interface as described in the Open Firmware standard. 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. The current implementation operates over a serial connection through the Power Macintosh modem port, with default settings as follows:

38400 baud
no parity
8 data bits
1 stop bit
XON/XOFF handshake
ANSI/VT102 terminal protocol

If the Open Firmware configuration variable `auto-boot?` is set to false, the Macintosh startup firmware will enter the user interface before initiating the startup process and will await input from the terminal connection; otherwise, the user interface can be invoked by holding the NMI switch on the motherboard closed immediately after system reset. The commonly used motion is to depress the NMI and Reset switches at the same time, and then release the Reset switch while holding the NMI switch closed until the user interface is invoked. This usually takes 1 to 3 seconds. When the user interface is invoked, it sends a bell character and a string identifying Open Firmware and its version number to the terminal.

The user interface operates as an interactive Forth environment, with necessary omissions and additions as appropriate to Open Firmware.

Here is a short list of commands available through the Open Firmware user interface. Note that several of them are combinations of commands that can be used separately.

`dev / ls` selects the root node and lists its children recursively, effectively dumping a name view of the device tree

`dev /bandit/gc .properties`
selects `gc` (the node representing the Grand Central ASIC, which controls many Macintosh I/O features) as the active package and displays its properties

`pwd` displays the pathname of the active package

`words` lists variables, constants, and methods of the active package (as in Forth, but in the scope of the current package only)

`printenv` lists current and default settings of Open Firmware configuration variables

`boot` proceeds with the startup process, using the currently chosen device

`FFC00000 100 dump`
: dumps $100 bytes from virtual address $FFC00000, if that address is currently mapped in

`dump-device-tree`
: lists properties and methods of all the device tree nodes

`init-nvram`
: resets values in non-volatile RAM to default values

`shut-down`
: flushes caches to RAM and powers down the machine

PART THREE

Native PCI Card Drivers

This part of *Designing PCI Cards and Drivers for Power Macintosh Computers* tells you how to design and write runtime PCI card drivers for the second generation of Power Macintosh computers. 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 PCI drivers for PowerPC Macintosh computers.
- Chapter 7, "Writing Native Drivers," gives you details of native driver design and coding.
- Chapter 8, "Macintosh Name Registry," describes the Mac OS data structure that stores device information extracted from the PCI device tree.
- Chapter 9, "Driver Services Library," details the general services that Mac OS provides for device drivers, including interrupt services.
- Chapter 10, "Expansion Bus Manager," discusses a collection of timing and other bus-specific system services available to native device drivers.
- Chapter 11, "Graphics Drivers," describes the calls serviced by typical display drivers.
- Chapter 12, "Network Drivers," describes the construction of a sample network driver.
- Chapter 13, "SCSI Drivers," describes the construction of a sample native SCSI Interface Module (SIM) compatible with SCSI Manager 4.3.

P A R T T H R E E

Native Driver Overview

This chapter presents an overview of the PCI driver environment and services, or *I/O architecture,* available in the Macintosh system software for the second generation of Power Macintosh computers. 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, INITs, and application code. The Device Manager is being changed; future releases of Mac OS will support older Device Manager operations only for drivers written in MC68000-family microprocessor code running in emulation mode.

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 System Services" on page 49.

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" beginning on page xxiii.

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. Meanwhile, Mac OS is undergoing fundamental changes that provide better memory protection, pre-emptive scheduling of tasks, and improved I/O support.

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 features:

- native PowerPC execution of all driver code
- support for PCI bus operations
- new Device Manager support for concurrent operations
- improved interrupt mechanisms
- new driver support services
- a Name Registry

Mac OS provides these features only for PCI native device drivers. Existing drivers written in code for MC68000-family microprocessors 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 Macintosh drivers to the native model described in this chapter. The model standardizes Macintosh driver design so that PCI and non-PCI device drivers can be written to a single specification. Drivers that conform to the new driver framework will work unchanged in future releases of Mac OS.

Terminology

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

- ***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.
- ***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 problem areas discussed in this chapter.
- ***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.
- ***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.
- ***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).
- ***Properties:*** Each piece of information associated with an entry in the Name Registry is called a *property.* For example, a `driver-description` 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" beginning on page 72.
- ***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 administrator:*** A family administrator is a high-level system component that communicates configuration information to a device, using whatever mechanism is appropriate. Configuration information may be known only to the user or may be stored in a file system, and it may not be available when an entry is first added to the Name Registry. A family administrator can communicate with a family expert, a driver, or the Name Registry to install and retrieve configuration information. Mac OS currently contains no family administrators; it may include them in the future or third parties may supply them.
- ***System Programming Interface (SPI):*** The SPI is the 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 an SPI routine in Mac OS that defers interrupt processing. Application-level software does not generally have access to the SPI. For more information about the Macintosh SPI for PCI cards, see Chapter 9, "Driver Services Library."
- ***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.
- ***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.***
- ***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 fully described in *Inside Macintosh: PowerPC System Software.*
- ***ROM-based drivers:*** ROM-based drivers are drivers that are stored in a PCI 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." PCI ROMs usually also contain native run-time drivers for Mac OS, stored as CFM fragments; they are described in Chapter 7, "Writing Native Drivers."
- ***Disk-based drivers:*** Disk-based drivers are drivers that are stored in the Macintosh 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.
- ***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 accelerator card.
- ***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. The concepts include:

- separation of application and system services
- common packaging of loadable software
- separate layers of driver implementations
- 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 System 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 second generation of Power Macintosh computers, Apple starts distinguishing between APIs and System Programming Interfaces (SPIs). The distinction must be made because programming contexts are becoming increasingly specialized as Mac OS evolves.

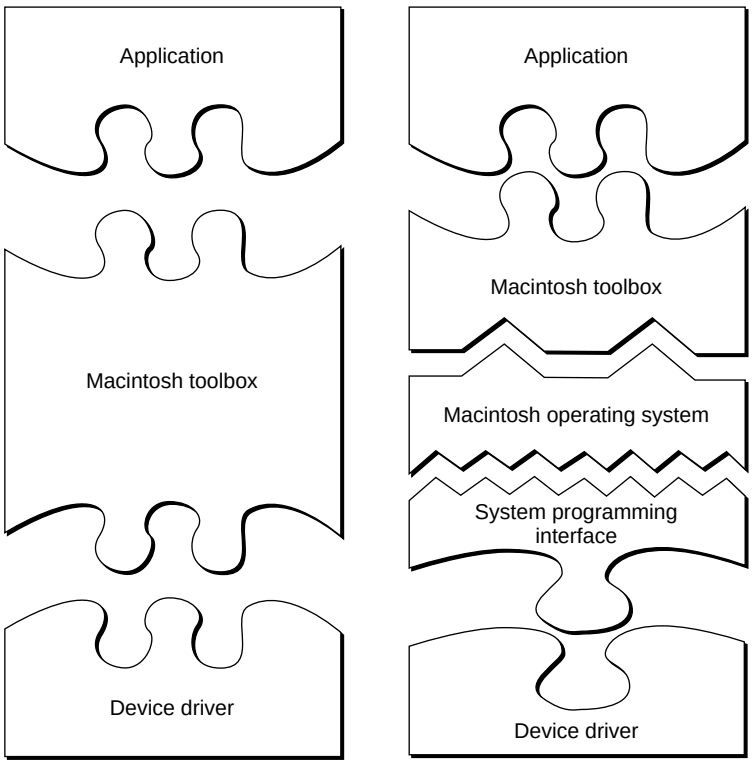
In present and future Mac OS releases, Toolbox services (for example, the `ModalDialog` function and Menu Manager calls) are not available to drivers. Drivers operate outside the user interface and the application software environment.

Note
Commands available through the concurrent Device Manager still constitute an API for generic drivers, as described in "Generic and Family Drivers" on page 55.

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

The separation of application and system services in Mac OS is a big change that starts with the second generation of Power Macintosh computers. The difference between the old API model and the new API/SPI model is diagrammed in Figure 6-1.

Figure 6-1 New system model



Common Packaging of Loadable Software

Native device drivers and SCSI Interface Modules (SIMs) are created as Code Fragment Manager (CFM) fragments. Each CFM fragment exports a driver description structure that the system uses to locate, load, and initialize the driver or SIM. Previously, device drivers were created as Macintosh resources.

Hence native drivers are packaged differently from previous Macintosh drivers. Because they are CFM fragments, they are allowed to have persistent global data storage in specific locations and they can be written in a high-level language without assembly-language headers. Each instance of a single driver or SIM has private static data and shares code with every other instance of that driver or SIM. 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 and SIMs as CFM code fragments is that a single device driver no longer controls multiple devices. Normally there is a driver instance for each device, although only one copy of the driver's code is loaded into memory.

Separate Layers of Driver Implementation

Native device drivers for the second generation of Power Macintosh computers can be divided conceptually into at least three sections: the main driver, the configuration section, and the control section.

The ***main driver*** (or lowest layer of the driver) is the code that resides in the operating system and responds to the I/O command set for a particular device or family. For example, generic device drivers (drivers of family type `'ndrv'`, discussed in "Generic and Family Drivers" on page 55) respond to the calls `Open()`, `Close()`, `Control()`, `Prime()`, and so on.

Main driver code should make no assumptions about any particular hardware settings or configuration. Any device configuration information (such as baud rate settings, the user's Ethernet address, the bit depth of the user's screen, and so on) should be obtained from the I/O command set as parameters passed to the main driver via `Init()`, `Open()`, or `Control()`. The main driver should not try to obtain device configuration information from the Resource Manager, File Manager, Slot Manager, PRAM utilities, or similar Toolbox services. These services will not be available to drivers under future releases of Mac OS, and using them will prevent drivers from being portable.

In the new native model, drivers are passive—they are driven by higher-level software. They are opened and controlled by clients that want to use their services. Drivers are not opened automatically or by themselves.

The ***configuration section*** of a native driver is a supporting piece of software that doesn't necessarily reside in the operating system. The configuration section can take many forms: it can be a resource of type `'INIT'` or a piece of user interface code in a resource of type `'CDEV'` or a device resource file of type `'RDEV'`. Its primary function is to exchange device configuration information with the device driver using `Init()`, `Open()`, `Control()`, and `Status()` calls. It has access to Toolbox APIs and system information that the main driver can't access itself. If a driver needs to do something at a high level, such as displaying a dialog box or reading resources from a file, it should do it in the configuration section.

IMPORTANT

PCI buses do not use hard-coded or calculated base addresses for card expansion ROM, RAM, and control registers. NuBus-based devices could calculate their base addresses from the number of the slot into which the card was inserted. PCI devices are located in memory as a result of the system configuration process, so main drivers and configuration sections must obtain base addresses from the Name Registry. ▲

The ***control section*** of a driver is the section that permits high-level software and user control. It communicates with the main driver by means of Device Manager control calls or software interrupts. Like the configuration section, the control section has full access to the Macintosh Toolbox.

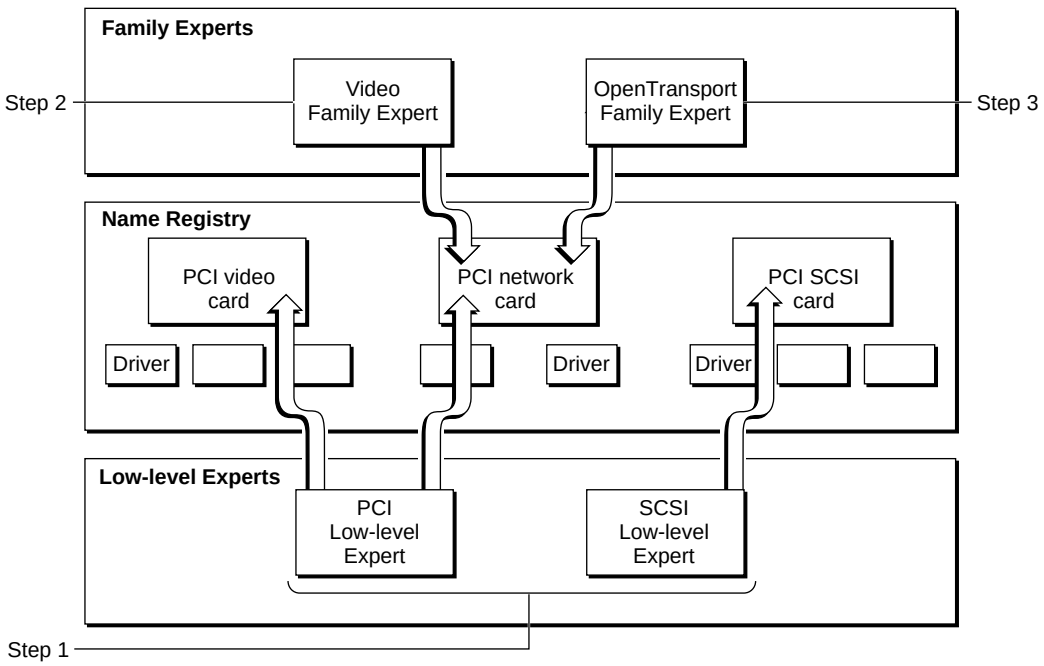
All drivers must have the main driver section, but not all will have configuration or control sections. The structure presented here is a development guide that shows how

code must be compartmentalized to match the environmental constraints that the native driver model places on different driver functions. Although main driver code can access only a simple set of services that provides for its basic needs, it gains the advantage of a fine level of control over system facilities such as paging and interrupts.

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 8.

Figure 6-2 Typical role of the Name Registry



Step 1: Low-level experts scxan for their device types and create Name Registry entries in which to deposit device drivers and properties.

Step 2: Family Expert locates all entries that match its service Category.

Step 3: Family Expert obtains driver and calls driver's initialization routine.

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 Macintosh ROM and in PCI card expansion ROMs install and update system information in the Registry.

- Other software in the startup process can then use the Registry to locate devices required to initialize the system.
- System firmware installs disk-based drivers and other system components in the Registry when the file system becomes available.
- Disk-based experts can then use information in the Registry to locate and initialize family devices.
- When device initialization driver code is called, the Registry provides configuration information for device drivers and family experts.

To make driver design easier, the Name Registry lets all types of device drivers be written identically, whether they are located in PCI 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 Registry allows for a comprehensive driver replacement capability, as described in "Finding, Initializing, and Replacing Drivers" beginning on page 120. All device entries and their corresponding code (drivers or SIMs) 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 SPI 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 Power Macintosh general-purpose I/O interconnection scheme because it allows the needs of each device family to be met independently of the needs of other families. The Name Registry helps PCI card developers group devices together and provide family services for those devices.

Mac OS for PCI-based Power Macintosh computers provides built-in support for device families such as the display family and the network family. Each of these 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 PCI 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 11, "Graphics Drivers."

A family expert such as the Open Transport expert interrogates the Registry for devices of a certain service category, verifying only that they are of the right category. For example, a software loopback device could appear in the Registry, the driver for which would take data from a source and return it back to the same source. To install a loopback Registry entry, the loopback configuration software would call the 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; there would be no bus associated with the loopback device. 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 PCI expansion ROMs. Disk-based drivers are located in the Macintosh 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 Mac OS releases starting with the second generation of Power Macintosh, different execution levels will have different restrictions. Noninterrupt level execution may make use of nearly any operating system or 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:

- ***Noninterrupt level:*** the 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 nest, secondary interrupt handlers always run serially.

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 is described in "Generic Driver Framework" beginning on page 56.

Most drivers are generic. However, 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. An example of this type of driver is a networking device driver for the Open Transport environment. Networking device drivers under Open Transport are STREAMS drivers that provide industry standard STREAMS/DLPI interfaces to the Macintosh system. They are discussed in Chapter 12, "Network Drivers."

Drivers controlled by family experts use Family Programming Interfaces (FPIs). FPIs are defined for each family, and are not accessible to Macintosh applications. Should an application discover an FPI and try to make an FPI call, the application is likely to fail. In the next release of Mac OS software, the application will probably crash with an access violation because the device driver services are in a different address space than Macintosh applications.

All drivers with family-private FPIs must export well-defined family FPI names for both FPI data and FPI 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. PCI device drivers and SIMs that provide private family interfaces need not provide an additional native driver interface to the Macintosh system.

As an example of a family interface, Open Transport requires a family data structure called `install_info` and an FPI function whose name is `GetOTInstallInfo`. The `install_info` structure is used by Open Transport to link STREAMS modules to STREAMS device drivers. The Open Transport family expert calls the device driver FPI `GetOTInstallInfo` function as part of the installation process for native drivers of the `'OTAN'` service category. See Chapter 12, "Network Drivers," for more details on Open Transport driver requirements.

Other family drivers are described in Chapters 11 and 13.

Note
Device drivers only need to provide one programming 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 "Driver Loader Library" beginning on page 102. It is based on the `driver-description` property associated with device entries in the Name Registry, described in Chapter 8.

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 "Finding, Initializing, and Replacing Drivers" beginning on page 120.

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 runtime drivers—that is, drivers of family type `'ndrv'`.

Device Manager

The traditional Macintosh ***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" beginning on page 69.

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 a PCI expansion ROM, or in the data fork of a Preferred Execution Format (PEF) file. File-based generic and family driver fragments have no 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 which describes the drivers type, functionality, and characteristics. This data structure is described in "Driver Description Structure" beginning on page 73.

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 `Open`, `Close`, `Read`, `Write`, `Control`, `Status`, `KillIO`, `Initialize`, and `Finalize` 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 PCI driver framework includes a ***Driver Services Library (DSL)*** that supplies the system programming interface (SPI) required by most generic drivers. SPIs are described in "Separation of Application and System Services" beginning on page 49. 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 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 Toolbox traps, low-memory globals, and similar vectors) will result in driver failure.

Converting Previous Macintosh Drivers

Previous Macintosh drivers have used standard ways of handling scheduling, memory management, interrupts, and configuration. Macintosh drivers have also had the freedom to call a set of services that are not available in the native driver model.

As mentioned in "Separation of Application and System Services" beginning on page 49, future releases of Mac OS will distinguish between Application Programming Interfaces (APIs) and System Programming Interfaces (SPIs). Services such as modal dialogs or Menu Manager calls will not be available to drivers; instead drivers will use only the interfaces provided by the Driver Services Library. Those parts of a driver that require services provided by the Macintosh Toolbox must be written to run at noninterrupt level, in the configuration or control sections described in "Separate Layers of Driver Implementation" beginning on page 51. These driver layers must communicate separately to the low-level driver code that directly controls the device hardware.

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" beginning on page 127 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.

Ensuring Future Compatibility

Several important environmental differences between the current release of Mac OS and future releases affect native drivers. Two of them are the following:

- Substantial changes in task 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 will 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 will 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 success with future releases.

Task and interrupt handling are discussed in detail in various sections of Chapters 7 through 9. Addressing problems are discussed next.

Note
The issues discussed here do not apply to classic 68K drivers, even though such drivers are also called via the Device Manager. 68K drivers are executed by Macintosh emulation software. ◆

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 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[11]` 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 be 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 the Mac OS, the Device Manager will copy only the `IOParamBlockRec` and `CntrlParam` structures as defined in *Inside Macintosh: Devices*.

Synchronous and Asynchronous Driver Operation

Due to tasking and addressing issues in future releases of Mac OS, data buffer handling will differ between synchronous and asynchronous calls. Synchronous calls to native drivers through the Device Manager will run on the thread of the calling application. This will allow direct accessibility to all data in `IOParamBlockRec` or `CntrlParam` structures and their associated data.

Asynchronous calls to native drivers will cause I/O operations within the device driver to run in a separate task context. This means that only that data which has been copied

or mapped by the Device Manager will be available to the native code that processes the I/O request.

One result of the different behavior of asynchronous and synchronous drivers is that the writer of a native driver must make careful implementation choices. The driver may be completely synchronous and do minimal data copying or mapping, but the application calling the driver will halt until the I/O request is complete. Alternatively, the driver may be completely asynchronous and concurrent. This will release application execution, but will require that all data be transferred in an `IOParamBlockRec` or `CntrlParam` structure, or via `PBRead` and `PBWrite` call buffers pointed to by the `ioBuffer` field of an `IOParamBlockRec`.

Another option is to support a mix of asynchronous and synchronous calls within the driver. This option is straightforward for nonconcurrent drivers, and is possible (with restrictions) for concurrent drivers. Mixing asynchronous and synchronous calls results in a more complex API, but may allow for special purpose optimizations. Nonconcurrent drivers use Device Manager queueing and expect to handle no more than one outstanding I/O request at a time. This mechanism lets the Device Manager handle address mapping or copying invisibly.

To support a mix of synchronous and asynchronous commands within a concurrent driver, the driver must ensure that `PBRead` and `PBWrite` calls are the only asynchronous calls. All other calls must by synchronous. Concurrent drivers supporting a mix of synchronous and asynchronous calls that result in queued I/O requests are not possible with the current version of Mac OS because the driver would have to be aware of task switching primitives that are not available. A concurrent driver that allows only synchronous control and status calls, and never queues these requests, can make use of data that is available through the `IOParamBlockRec`.

Sharing Data With Applications

A concurrent or nonconcurrent driver wishing to share a data buffer with an application should do the following. The application should issue an asynchronous read or write command to the driver supplying the data buffer address and byte count in the `ioBuffer` and `ioReqCount` fields in the `IOParamBlockRec`.

To indicate to the Device Manager that the `ioBuffer` to be shared must be mapped (not copied), the `ioMapBuffer` flag must be set in the `ioPosMode` field of the `IOParamBlockRec` structure. The driver and the application can share the buffer for the duration of the asynchronous call. When sharing is complete, the driver should complete the asynchronous call using the `IOCommandIsComplete` service described on page 69.

Note
The issues discussed here are separate from the concurrent programming issues and requirements discussed in "Concurrent Generic Drivers" beginning on page 68. The addressing issues detailed here affect only the movement of data between applications and device drivers. ◆

Summary

The I/O architecture that is defined in this chapter sets a durable standard for writing Macintosh device drivers. This standard will be supported in future releases of Mac OS, and device drivers that conform to it will 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 below.

Divide Drivers Into Sections

The main section of a device driver is a low-level piece of operating system software. Part of the task of allowing a driver to be portable in a modern operating system is to determine which portions of the driver belong in the operating system and which do not, as discussed in "Separate Layers of Driver Implementation" beginning on page 51. Remember that code that resides in the operating system gains the advantage of fine control over system facilities like paging or interrupts at the cost of giving up access to the rich set of high-level APIs available via the Macintosh Toolbox.

Use the System Programming Interfaces

The use of the System Programming Interfaces is essential to a driver's portability in future Mac OS releases. These are the set of programming interfaces for device drivers that are guaranteed to be common across operating system releases. When writing the main driver section of a device driver, never make Toolbox API calls. Instead, use the corresponding System Programming Interfaces. These sets of calls let you deal more naturally with device driver issues than the Toolbox API does, because it was intended for applications. If you find that functionality you depended on in the Toolbox has been removed from the set of SPIs provided, perform that function in the configuration or control sections of the 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. The Name Registry is key to implementing several important features in the PCI-based Power Macintosh I/O architecture:

- ***Effective driver replacement and overloading capability.*** This allows you, or third parties, to release updates to drivers that shipped with bugs.
- ***Dynamic driver loading and unloading.*** The Name Registry provides a dynamic and flexible environment for identifying devices. This capability will be necessary for supporting devices such as hot-swappable PCMCIA cards.

- ***Simplification of driver writing.*** Programmers no longer follow different rules for writing device drivers located on the main logic board, NuBus, the PCI bus, or the PCMCIA bus.
- ***Making device drivers more hardware-independent.*** The Name Registry provides the layer of abstraction necessary for driver writers to remove conflicting device identification and device information. These conflicts, which occurred previously with the Slot Manager, prevented drivers from being portable to new versions of Macintosh hardware.

For further information about the Name Registry, see Chapter 8, "Macintosh Name Registry."

Writing Native Drivers

This chapter tells you about Macintosh native runtime 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 driver code from the MC68000 environment to the PowerPC 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" beginning on page 56. ◆

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" beginning on page 55. The general characteristics of both kinds of native drivers are briefly summarized in the next sections.

Native Container Format

The container format for native PowerPC device drivers is the container format supported by the Code Fragment Manager. The CFM format provides all mechanisms necessary for drivers, is integrated with Mac OS, and is documented in *Inside Macintosh: PowerPC System Software.*

Previous device drivers for use with MC68000-family microprocessors were located in `'DRVR'` resources, as described in *Inside Macintosh: Devices.*

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`, is exported through the CFM's symbol mechanism. It is documented in "Driver Description Structure" beginning on page 73.

Driver Description information helps match drivers with devices. It also lets the Device Manager pick the best driver among multiple candidates. For example, it lets a newer driver on disk override a ROM-based driver.

Native Driver Code Exports

Previous Macintosh drivers exported five callable routines: `Open`, `Close`, `Prime`, `Control`, and `Status`. 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 (`Initialize`, `Finalize`, `Open`, `Close`, `Read`, `Write`, `Control`, `Status`, `KillIO`, `Replace`, or `Superseded`) and command kind (`Synchronous`, `Asynchronous`, or `Immediate`). `DoDriverIO` is described in "DoDriverIO Entry Point" beginning on page 78.

Native Driver Imports

The CFM requires that fragment imports be identified in some manner. With generic drivers, this is done by linking the device driver fragment code to the Driver Services Library; family drivers may also be linked to Family Libraries. The linking lets the fragment's symbols be bound at execution 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" beginning on page xxiii. ▲

In the past, driver imports have not been rigidly characterized, although most device driver writers have discovered what works and what does not. 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" beginning on page 57. See also the family-specific information in Chapters 11, 12, and 13.

Drivers for Multiple Cards

Drivers that support more than one PCI expansion card (or multiple sections on one card) should use the Code Fragment Manager to import a shared library for both code and data. The CFM links instances of the native driver on the fly when the driver is loaded by the Driver Loader Library. Follow these design guidelines:

- Put the shared library in the Extensions folder along with the driver.
- 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.
- Load the driver multiple times with the functions `InstallDriverFromDisk` or `InstallDriverFromFile`, passing the `RegEntryID` of each device as a parameter. Instances of the driver for each card will be installed in the unit table with different `RefNum`s.

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 Macintosh system software 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.*

The Device Manager has traditionally been the gateway for device drivers to use the Macintosh Toolbox. For drivers compatible with the MC68000 microprocessor family (*68K* drivers), the Device Manager's capabilities and services remain unchanged. For generic drivers compatible with the PowerPC microprocessor, 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 generic native device drivers, see "Writing a Generic Device Driver," later in this chapter. If you are already familiar with writing device drivers in the MC68000 environment, using former versions of the Device Manager, 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 runtime 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. `Initialize`, `Finalize`, `Open`, `Close`, `KillIO`, `Replace`, and `Superseded` 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 `Open` and `Close` calls, which connect the driver independently of initialization and finalization. In the past, the first (and only) `Open` and `Close` 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 permissions are violated.
- The existing Device Manager processes zero-length reads and writes on behalf of the driver. New drivers must accept zero-length read and write commands and respond to them 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 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`.
- 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. This reentrancy requirement still holds for the `ioDone` call in 68K driver code.

- 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.
- A native device driver cannot make use of the `dCtlEMask` and `dCtlMenu` fields of its driver control block.
- Native drivers cannot be used for creating desk accessories.
- Native drivers must use only those services provided by the Driver Services Library or Family Libraries.

Native Driver Limitations

The ability of Mac OS to support generic native drivers does not mean that there is a native I/O subsystem in the second generation of Power Macintosh computers or that mid-68K instruction interrupt handling is possible. Moreover, 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 either a native or a "fat" version of the Device Manager.

Additionally, 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 to on only one request at a time. Although multiple requests could be pending for a driver, the Device Manager passed each new request only when the it was certain that the driver was idle.

Many clients of the present Device Manager contain workarounds that let the driver handle multiple requests concurrently. The Device Manager now lets native PowerPC 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. In concurrent mode, 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 queing 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

Besides the `IODone` routine and its associated vector `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 IOPB 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 IOPB'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 IOPB `result` field. The driver should never try to modify `result`.

IOCommandIsComplete

`IOCommandIsComplete` tells the Device Manager that an I/O request has been completed.

```
OSStatus IOCommandIsComplete  (CommandID     Command,
                               OSStatus      Results);
```

`Command` Specifies the ID of a command.

`Results` Returns the status value to place in the IOPB

DESCRIPTION

The parameter `Command` specifies the ID of a command being completed. Note that the value of this ID is dependent on the operating system version. The parameter `Results` specifies the status value to place in the IOPB. The driver must assure that the request that corresponds to `Command` is not queued internally when `IOCommandIsComplete` is called. The driver must not access the parameter block after `IOCommandIsComplete`.

RESULT CODES

`noErr`	0	No error

Passes return code from I/O operation

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 IOPB) is passed to the Device Manager.
3. The Device Manager uses the `refNum` in the IOPB 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 IOPB 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 post the request to a privately-managed queued.
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 primary or secondary interrupt routine) that the device I/O action has finished. At this time, the driver scans its private queue looking for the IOPB representing the I/O action.
10. The driver uses the IOPB `commandID` stored at (`ThePb` -> `ioParam.ioCmdAddr`) to issue an `IOCommandIsComplete` call. Drivers using the `kdriverQueuesIOPB` bit must make sure the IOPB is not on any queue when calling `IOCommandIsComplete`.
11. The Device Manager places the result in the IOPB.
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.
13. Control returns to the driver. The driver should not attempt to access the IOPB 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

Future versions of Mac OS will enforce strict runtime execution limitations based upon execution contexts. Considerable effort has been spent on normalizing these contexts to ensure that high-level language software can run directly with no interface glue. The environments in which code execution can occur are described in "Noninterrupt and Interrupt-Level Execution" beginning on page 54 and may be summarized as follows:

- ***Noninterrupt 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 in System 7. The secondary interrupt queue is filled with requests to execute subroutines posted for execution by hardware interrupt handlers. Unlike hardware interrupt handlers, which can nest, secondary interrupt handlers always execute sequentially. For synchronization purposes, code running at noninterrupt level may also post secondary interrupt handlers for execution; these are processed synchronously from the perspective of the noninterrupt level, but are serialized with all other secondary interrupt handlers.

Different execution levels have different restrictions. Noninterrupt 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.

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. ◆

Driver Execution

The System 7 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 System 7 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.

A driver's `Open`, `Close`, `Initialize`, `Finalize`, `Replace`, and `Superseded` entry points are always invoked at noninterrupt level. This is the only opportunity that a driver has to allocate memory or use other services that are only available at the noninterrupt level.

"Service Limitations" beginning on page 216 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. But if your product is a specialized device that can only be used by one application, it may be easier to control the device using private code in the application.

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
Generic drivers alone interact with the Device Manager. The only part of this section that applies to family drivers is "Driver Description Structure" beginning on page 73.

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 which 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 a PCI 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 Macintosh file system ignores the file's creator; by specifying a custom creator value assigned by Apple, you can use this value to distinguish drivers. For a discussion of this technique, see "Using NVRAM to Store Device Properties" beginning on page 222.

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 `'ndrv'` driver type requires that the CFM package export a single code entry point, `DoDriverIO`, which passes all driver I/O requests. `DoDriverIO` must respond to the `Open`, `Close`, `Read`, `Write`, `Control`, `Status`, `KillIO`, `Initialize`, `Finalize`, `Replace`, and `Superseded` 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.

Driver Description Structure

The structure `DriverDescription` is used to match drivers with devices, set up and maintain a driver's runtime 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 of
                                        DriverDescription*/
};
```

```
typedef UInt32 DriverDescVersion;
enum {
   kInitialDriverDescriptor = 0 /*Version 1 of DriverDescription*/
};
```

Field descriptions

`driverDescSignature`
Signature of this `DriverDescription` structure; currently `'mtej'`.

`driverDescVersion`
Version of this Driver Description structure, used to distinguish different versions of `DriverDescription` that have the same `driverDescSignature`.

`driverType` Structure that contains driver identification information, such as vendor, device, revision IDs, driver name, class code, and version.

`driverOSRuntimeInfo`
Structure that contains driver runtime 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 7-1.

Listing 7-1 Typical Driver Description

```
DriverDescription TheDriverDescription =
{
   // Signature Info
      kTheDescriptionSignature,        // signature always first
      kInitialDriverDescriptor,        // version second

   // Type Info
      {
         "\pAAPL,viper",               // Our device's Name (must match)
         1, 0, 0x80, 0,                // Major, Minor, Stage, Rev
      },

   // OS Runtime Info
      {
         kdriverIsUnderExpertControl   // Runtime Options
       + kdriverIsOpenedUponLoad,
         "\p.WeitekVideo",
      },
```

```
   // OS Service Info
      {
         2,                            // Number of Service Categories
         {
            {
               kServiceCategoryDisplay,// We support the display category
               kndrvType1,             // Type1
               { 1, 0, 0, 0 }          // Major, Minor, Stage, Rev
            },
            {
               kServiceCategoryndrvdriver, // We also support 'ndrv' category
               kndrvType1,             // Type1
               { 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" beginning on page 122.

```
struct DriverType {
   Str31             nameInfoStr;
   NumVersion        version;
}

typedef UInt32    DeviceTypeMember;
typedef struct    DriverType DriverType;
typedef struct    DriverType *DriverTypePtr;
```

Field descriptions

`nameInfoStr` Driver 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 PCI `name` property in the Name Registry. For further information, see "Standard Properties" beginning on page 156.

`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 on disk.

Driver Runtime Structure

The `DriverOSRuntime` structure contains information that controls how the driver is used at run time.

```
struct DriverOSRuntime {
   RuntimeOptions    driverRuntime;
   Str15             driverName;
   UInt32            driverDescReserved[8];
};

typedef OptionBits RuntimeOptions;
typedef struct DriverOSRuntime DriverOSRuntime;

typedef struct DriverOSRuntime *DriverOSRuntimePtr;
enum {                            /*driverRuntime bit constants*/
   kdriverIsLoadedUponDiscovery  = 1,  /*auto-load driver when
                                         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             = 10  /*Device Manager doesn't
                                         queue IOPB*/
};
```

Field descriptions

`driverRuntime` Options used to determine runtime 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 does not queue the IOPB to the DCE request before calling the driver.

`driverName` Driver name used by Mac OS if driver type is `'ndrv'`. Mac OS copies this name to the name field of the DCE. This field is unused for other driver types.

`driverDescReserved`
Reserved for future use.

Driver Services Structure

The `DriverOSService` structure describes the services supported by the driver that are available to applications and 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 serviceCatagory*/
   kServiceCategoryDisplay = 'disp',      /*display*/
   kServiceCategoryopentransport ='otan', /*open transport*/
   kServiceCategoryblockstorage = 'blok', /*block storage*/
   kServiceCategorySCSISim = 'scsi',      /*SCSI device*/
   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:
`'disp'`	video display family
`'otan'`	Open Transport
`'blok'`	block drivers family
`'ndrv'`	generic native driver devices

`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 `Open`, `Close`, `Read`, `Write`, `Control`, `Status`, `KillIO`, `Initialize`, `Finalize`, `Replace`, and `Superseded` commands.

```
OSErr DoDriverIO  (AddressSpaceID      spaceID
                   IOCommandID         ID,
                   IOCommandContents   contents,
                   IOCommandCode       code,
                   IOCommandKind       kind);

typedef KernelID AddressSpaceID;
```

`spaceID` ID of address space

`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, 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 UInt32 IOCommandID;
enum{
   kInvalidID = 0
};

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 7-2.

Listing 7-2 Driver handler for DoDriverIO

```
OSErr
DoDriverIO( AddressSpaceID      SpaceID,
            IOCommandID         theID,
            IOCommandContents   theContents,
            IOCommandCode       theCode,
            IOCommandKind       theKind )
{
   OSErr   ImmediateResult;
   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 ( (theKind & kImmediateIOCommandKind) != 0 )
        return result;

    // else return noErr via the new IODone
    return IOCommandIsComplete( theID, result );
}
```

Getting Command Information

Any command in progress that the Device Manager has sent to a native driver can be examined using `GetIOCommandInfo`.

GetIOCommandInfo

`GetIOCommandInfo` returns information about an active native driver I/O command.

```
OSErr GetIOCommandInfo  (IOCommandID         theID,
                         IOCommandContents   *theContents,
                         IOCommandCode       *theCommand,
                         IOCommandKind       *theKind);
```

`theID`	Command ID
`theContents`	Pointer to the IOPB or Initialize/Finalize contents
`theCommand`	Command code
`theKind`	Command kind (synchronous, asynchronous, or immediate)

The `GetIOCommandInfo` command will not work after a driver has completed a request.

Responding to Device Manager Requests

Native drivers respond to Device Manager requests by handling a single call, `doDriverIO`. 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.

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 `Initialize` and `Finalize` commands to a native driver as its first and last commands. The `Initialize` command gives the driver startup information; the `Finalize` command informs the driver that the system would like to unload it. `Open` and `Close` actions are now separate from initialization and finalization; in the past, `Open` and `Close` calls were used as the initialization and finalization mechanism.

A typical framework for a generic driver handler for Device Manager finalization and CFM initialization and termination commands is shown in Listing 7-3.

Listing 7-3 Initialization, Finalization, and Termination handlers

```
refNum      MyReferenceNumber;
RegEntryID  MyDeviceID;

OSErr DoInitializeCommand
         ( refNum myRefNum, regEntryIDPtr myDevice )
{
   // Remember our refNum and Registry Entry Spec
   MyReferenceNumber = myRefNum;
   MyDeviceID = *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 first check the device's `AAPL,address` property to see that needed resources have been allocated. The `AAPL,address` property is described in "Accessing Device Registers" beginning on page 235.

The initialization code should also allocate any private storage the driver requires and place a pointer to it in the static data area that the Code Fragment Manager provides for each instance of the driver. 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,MAacOS,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 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" beginning on page 122. PCI `vendor-id` and `device-id` properties are discussed in "Finding, Initializing, and Replacing Drivers" beginning on page 120.

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 cancelling 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" beginning on page 190.

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 Macintosh system software 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 routines are immediate.

Typical code for keeping track of `open` and close commands is shown in Listing 7-4.

Listing 7-4 Managing open and close commands

```
long myOpenCount;

OSErr DoOpenCommand (ParmBlkPtr thePb)
{
   myOpenCount++;
   return noErr;
}

OSErr DoCloseCommand (ParmBlkPtr thePb)
{
   myOpenCount--;
   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" on page 88 discusses how to complete an asynchronous read or write routine.

Listing 7-5 shows a sample read routine.

Listing 7-5 Example driver read/write routine

```
short myLastErr;     /* Globals */
long  myLastCount;

OSErr DoReadCommand (IOpb pb)
{
long  numBytes;
short myErr;

numbytes = pb -> IORegCount;
   {
      /* do the read into pb -> iobuffer */
   }
myLastErr = myErr;         /* store in globals */
```

```
myLastCount = myLastCount;
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.

Listing 7-6 shows a sample control routine, `DoControlCommand`.

Listing 7-6 Example driver control routine

```
MyDriverGlobalsPtr   dStore;

OSErr DoControlCommand (ParamBlkPtr pb)
{
   switch (pb->csCode)
   {
      case kClearAll:
         dStore->byteCount = 0;
         dStore->lastErr = 0;
         return(noErr);
      default: /* always return controlErr for unknown csCode */
         return(controlErr);
   }
}
```

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.

Note
An IODone call with a status code of `PBBusy` causes a fatal error. ◆

Listing 7-7 shows a sample status routine, `DoStatusCommand`.

Listing 7-7 Example driver status routine

```
MyDriverGlobalsPtr   dStore;

OSErr DoStatusCommand (ParamBlkPtr pb)
{
   switch (pb->csCode)
   {
      case kByteCount:
         pb->csParam[0] = dStore->byteCount;
         return(noErr);
      case kLastErr:
         pb->csParam[0] = dStore->lastErr;
         return(noErr);
      default: /* always return statusErr for unknown csCode */
         return(statusErr);
   }
}
```

The control routine must return `controlErr` for any `csCode` values that are not supported. You can define driver-specific `csCode` values if necessary, as long as they are within the range $80 through $7FFF.

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 Err;
```

`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 (for example, 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. The driver must return control to the Device Manager by calling `IOCommandIsComplete`.

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 working.

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 begin assume control of the device. The command contents are the same as a `kInitializeCommand`. The incoming driver should take the following actions:

- Retrieve the state stored in the Name Registry.
- 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:

- ***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" beginning on page 190.

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 two 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 the expansion ROM of a PCI card, 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 inside the System folder.

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 "Finding, Initializing, and Replacing Drivers" beginning on page 120 for driver loading and installation details. "Driver Loader Library" beginning on page 102 provides details of the mechanisms available for installing and removing drivers that are listed in the Device Manager Unit Table.

Table 7-1 lists the driver unit numbers that are reserved for specific purposes.

Table 7-1 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 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 communicated between the family subsystem manager 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 which SCSI device to boot from. The information communicated back to the Start Manger is family-specific (specific to the SCSI Mgr) and contains necessary data for system startup— SCSI bus ID, device ID, and disk partition. Each I/O subsystem defines its own 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 Macintosh gestalt mechanism, see *Inside Macintosh: Operating System Utilities.* This book is described in "Apple Publications" beginning on page xxiii. The primary differences between driver gestalt and the traditional Macintosh gestalt mechanism are that driver gestalt has no `NewGestalt` or `ReplaceGestalt` functionality and information is provided independently for each driver.

System gestalt for PCI-based Macintosh computers, which is different from driver gestalt, is described in "Macintosh System Gestalt" beginning on page 236.

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

DriverGestaltIsOn

`DriverGestaltIsOn` lets other code test whether or not a driver supports driver gestalt.

```
Boolean DriverGestaltIsOn (DriverFlags flags);
```

`flags` `flags` word in driver's DCE

DESCRIPTION

`DriverGestaltIsOn` returns `true` if bit 2 in the DCE flags word is set, `false` otherwise.

RESULT CODES

`noErr`	0	No error

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 `csCodes` with a `statusErr` value, and to all unsupported control `csCodes` with a `controlErr` value. This rule is the most important for drivers to follow after calling `DriverGestaltOn`.

- It should be capable of closing properly, 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 7-2 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` return. The `kcsDriverGestalt` and `kcsDriverConfigure` codes produce the principal new functionality of the native driver model. For historical reasons, setting the `DriverGestaltEnable` bit also requires that the other calls listed in Table 7-2 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 7-2 Driver gestalt codes

Name	Value	Description
Status codes		
`kcsDriverGestalt`	43	General status information
`kcsGetPartitionStatus`	50	Gets status of partition
`kcsGetPartitionInfo`	51	Gets a partition information record
`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
`kcsRegisterPartition`	50	PC Exchange needs a new drive for a non-Mac OS partition
`kcsGetADrive`	51	Asks driver to create a new drive
`kcsProhibitMounting`	52	Prohibit mounting the listed drives
`kcsSetPowerMode`	70	Sets card power mode*

* For a discussion of power modes, see "Card Power Controls" beginning on page 235.

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 `flags` word is the following:

Name	Value	Description
`kbIsAppleTalk`	0	
`kbDriverGestaltEnable`	2	Supports driver gestalt
`kbIsNDev`	3	
`kbIsConcurrent`	4	
`kbIsOpen`	5	
`kbIsRamBased`	6	(Not used with native drivers)
`kbIsActive`	7	
`kbReadEnable`	8	
`kbWriteEnable`	9	
`kbControlEnable`	10	
`kbStatusEnable`	11	
`kbNeedsGoodbye`	12	
`kbNeedsTime`	13	
`kbNeedsLock`	14	
`kmIsNdrvMask`	1	`kbIsNdrvMask`
`kmIsConcurrentMask`	1	`kbIsConcurrentMask`
`kmIsAppleTalkMask`	1	`kbIsAppleTalk`
`kmIsAgentMask`	1	`kbIsAgent`
`kmDriverGestaltEnableMask`	1	`kbDriverGestaltEnable`
`kmIsOpenMask`	1	`kbIsOpen`
`kmIsRamBasedMask`	1	`kbIsRamBased`
`kmIsActiveMask`	1	`kbIsActive`
`kmReadEnableMask`	1	`kbReadEnable`
`kmWriteEnableMask`	1	`kbWriteEnable`
`kmControlEnableMask`	1	`kbControlEnable`
`kmStatusEnableMask`	1	`kbStatusEnable`
`kmNeedsGoodbyeMask`	1	`kbNeedsGoodbye`
`kmNeedsTimeMask`	1	`kbNeedsTime`
`kmNeedsLockMask`	1	`kbNeedsLock`

Using DriverGestalt and DriverConfigure

Status `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` 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;
   IOCompletionUPP      ioCompletion;
   OSErr                ioResult;
   StringPtr            ioNamePtr;
   short                ioVRefNum;
   short                ioCRefNum;  /* refNum for I/O operation*/
   short                csCode;     /* == driverGestaltCode */
   OSType               driverGestaltSelector;
   DriverGestaltInfo    driverGestaltResponse;
};

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;
   DriverGestaltInfo    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 7-3.

Table 7-3 DriverGestalt selectors

Selector	Description	Response type
`'vers'`	The version number of the driver	`NumVersion`*
`'devt'`	Type of device the driver is driving	`DevTResponse`
`'intf'`	Immediate location (or interface) for device	`IntfResponse`
`'sync'`	True if driver only behaves synchronously	`SyncResponse`
`'wide'`	True if driver supports the `ioWPosOffset` for 64-bit addressing	`WideResponse`
`'boot'`	Parameter RAM value to designate this driver/drive	`BootResponse`
`'purg'`	True if driver has purge permission	`Boolean`
`'lpwr'`	True if driver supports power switching	`Boolean`
`'psup'`	True if driver supports power calls	`Boolean`
`'psta'`	True if device is currently in high power mode	`Boolean`
`'pmx5'`	Maximum power consuption at 5 volts	
`'pmn5'`	Minimum power consuption at 5 volts	
`'pmx3'`	Maximum power consuption at 3.3 volts	
`'pmn5'`	Minimum power consuption at 3.3 volts	
`'getV'`	Software version information for PC Exchange†	`GetVResponse`
`'getL'`	List of device IDs for PC Exchange	`GetLResponse`
`'dAPI'`	Driver API information for PC Exchange	`DAPIResponse`

* The `NumVersion` data structure is described on page 115.
† This selector is obsolete; new drivers should use `'vers'`

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. ◆

The following response buffers are defined for some of the driver gestalt selectors listed in Table 7-3:

```
struct DriverGestaltSyncResponse
{
   Boolean behavesSynchronously;
};

struct DriverGestaltBootResponse
{
   unsigned char extDev;   /*  Packed target (upper 5 bits) LUN
                               (lower 3 bits) */
   unsigned char partition;/*  Partition */
   unsigned char slot;     /*  Slot */
   unsigned char sRSRC;    /*  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 locationType;
};
enum {
   kdgScsiIntf       = 'scsi',
   kdgPcmciaIntf     = 'pcmc',
   kdgIdeIntf        = 'ide ',
   kdgFireWireIntf   = 'fire',
   kdgExtBus         = 'card'
};
```

```
struct DriverGestaltGetVResponse {
   unsigned long  versionInfo;   /* vendor specific */
   char           *vendorName;   /* Pointer to C string for unique
                                    vendor id */
};

struct DeviceInfoRecord {
   struct DeviceInfoRecord *nextInfo;
   DeviceIdent             deviceID;
   short                   identifier; /* to be used as a unique
                                          identifier */
};

struct DriverGestaltGetLResponse
{
   DeviceInfoRecordPtr firstDeviceInfo;
};

struct DriverGestaltDAPIResponse
{
   short partitionCmds; /* see comment below*/
   short unused1;       /* remaining fields are zero */
   short unused2;
   short unused3;
   short unused4;
   short unused5;
   short unused6;
   short unused7;
   short unused8;
   short unused9;
   short unused10;
};

/* if bit 0 of partitionCmds is nonzero, the driver supports the
   following partition control and status calls:
      prohibitMounting (control, kProhibitMounting)
      partitionToVRef (status, kGetPartitionStatus)
      getPartitionInfo (status, kGetPartitionInfo)         */

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 the `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.

DriverConfigure Selectors

No `DriverConfigure` selectors are currently defined; however, the control call with `csCode` = 43 will be used in the future to add driver control functions. Drivers setting the `DriverGestaltEnable` bit should not implement this control call for other uses. To use the `DriverConfigure` call use the `driverConfigureSelector` field to choose an operation and pass parameters to it with the `driverConfigureParameter` field. Multiple parameters can be passed by means of a pointer to a structure.

Other Control and Status Calls

This section discusses how native drivers should respond to driver gestalt control and status calls other than `DriverConfigure` and `DriverGestalt`—that is, calls with `csCode` values other than 43.

SetStartupDrive Control Call

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 which control elements in the drive queue) that set the `driverGestaltEnable` bit must implement this control call or return `controlErr`.

PC Exchange Support Calls

This section discusses driver gestalt calls that support the Macintosh PC Exchange system extension. Mass storage drivers (those which control elements in the drive queue) that set the `driverGestaltEnable` bit must implement these calls or return `controlErr`.

RegisterPartition Control Call

The `RegisterPartition` control call (`csCode` = 50) registers a non-Macintosh partition found on a disk. The driver should fill in `csParam` as follows:

```
(long *)csParam[0]  <- DrvQElPtr /* DrvQEl of partition */
(long *)csParam[1]  <- /* Start of partition in logical blocks */
(long *)csParam[2]  <- /* Size of partition in logical blocks */
```

GetADrive Control Call

The `GetADrive` control call (`csCode` = 51) asks the driver to get a drive. There are no parameters passed into `GetADrive`, but it must return the `DrvQElPtr` for the drive in `csParam[0]`.

ProhibitMounting Control Call

The `ProhibitMounting` control call (`csCode` = 52) prevents the mounting of a partition. The `csParam[0]` 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;
```

GetPartitionStatus Status Call

The `GetPartitionStatus` status call (`csCode` = 50) retrieves the status of a partition. The driver should fill out `csParam` as follows:

```
(long *)csParam[0]   <-    /* partInfoRecPtr for partition */
(short *)csParam[1]  <-    /* address of a short for response */
```

The variable pointed to by `csParam` must be filled with the `VRefNum` for a volume mounted on the partition. If none exist, the driver must return 0.

GetPartitionInfo Status Call

The `GetPartitionInfo` status call (`csCode` = 51) returns information about a partition in the `partInfoRec` structure described above in "ProhibitMounting Control Call." The `csParam[0]` field contains a pointer to an empty `partInfoRec` structure, which the driver fills out 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
```

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 which 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.

Drivers which support low power mode calls should return true to the `'lpwr' DriverGestalt` call listed in Table 7-3 on page 95. 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 Status Call

A status call with a `csCode` of 120 returns the `DeviceIdent` for the primary SCSI device being controlled by a driver. SCSI drivers that set the `driverGestaltEnable` bit must implement this `csCode` as described or return `statusErr`.

GetCDDeviceInfo Status Call

A status call with a `csCode` 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 which set the `driverGestaltEnable` bit must either implement this csCode or return `statusErr`.

```
struct CDDeviceCharacteristics
{
   unsigned char  speedMajor; /* High byte of fixed point number
                                 for drive speed */
   unsigned char  speedMinor; /* Low byte of "" CD 300 == 2.2,
                                 CD_SC == 1.0 etc. */
   unsigned short cdFlags;    /* Flags for features of drive */
};
```

```
enum                /* Flags for CD Features field (cdFlags) */
{
   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
};
```

Driver Loader Library

This section describes the ***Driver Loader Library***, a CFM shared-library extension to the Macintosh Device Manager. The Driver Loader Library provides services to locate, install, and remove generic drivers. 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 which belong to a well-defined family type (such as networking devices within OpenTransport) 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 four major functions for drivers:

- loading
- installation in the Unit Table
- removal from the Unit Table
- information about installed drivers

Figure 7-1 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 7-1 Position of Driver Loader Library

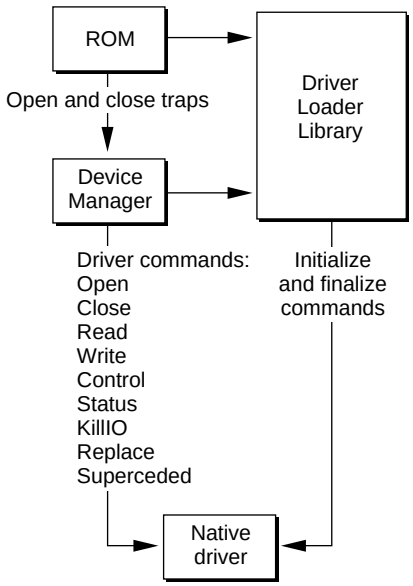
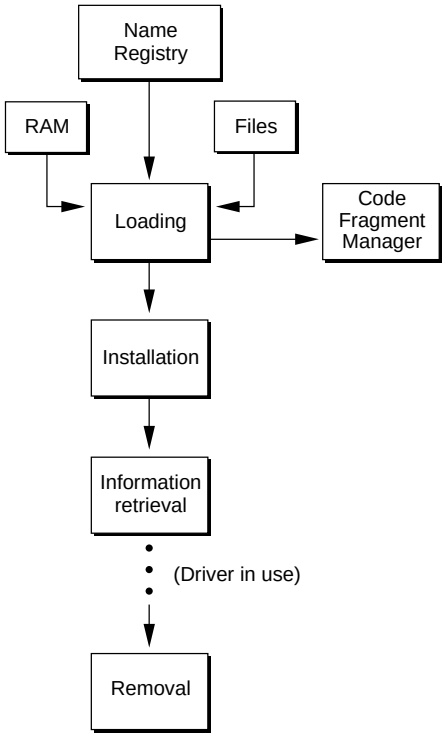


Figure 7-2 shows the relationship of the Loader Library's four functionality groups:

Figure 7-2 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
- `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

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
- a pointer to the `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.

GetDriverMemoryFragment

`GetDriverMemoryFragment` loads a code fragment driver from an area of memory.

```
OSErr GetDriverMemoryFragment
         (Ptr                    memAddr,
          long                   length,
          Str63                  fragName,
          ConnectionID           *fragmentConnID,
          NuDriverEntryPointPtr  *fragmentMain,
          DriverDescriptionPtr   *theDriverDesc);
```

`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`)
`theDriverDesc`	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 `theDriverDesc`.

RESULT CODES

`noErr`	0	No error
All CFM errors (See *Inside Macintosh: PowerPC System Software*)		

GetDriverDiskFragment

`GetDriverDiskFragment` loads a code fragment driver from a file using the CFM search path.

```
OSErr GetDriverDiskFragment
         (FSSpecPtr              theFragmentSpec,
          ConnectionID           *fragmentConnID,
          NuDriverEntryPointPtr  *fragmentMain,
          DriverDescriptionPtr   theDriverDesc);
```

`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 file system specification, `GetDriverDiskFragment` uses the CFM search path 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 `theDriverDesc`.

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 or from a device tree property that represents the "best" driver for a device—that is, the latest version of the most appropriate driver. The algorithm for determining the best driver is described in "Matching Drivers With Devices" beginning on page 122.

```
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 CFM 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`.

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" beginning on page 122.

```
OSErr GetDriverForDevice(RegEntryIDPtr          device,
                         ConnectionID           *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 `theDriverDesc`.

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found

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

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`.

The installing software can specify a desired range of unit numbers in the Unit Table. For example, SCSI drivers use the range 32 to 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 as well.

Note
The unit table may grow only if the unit number is between 48 and the current highest unit number as returned by the `HighestUnitNumber` routine described on page 117. It cannot grow if it has reached its maximum size. ◆

When installing a native driver, the caller also passes the `RegEntryIDPtr` of the device which the driver is to manage. This pointer (along with the `refNum`) 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
         (ConnectionID           fragmentConnID,
          NuDriverEntryPointPtr  *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
         (ConnectionID     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`

DESCRIPTION

`InstallDriverFromFragment` installs a driver that is located in a CFM code fragment, using the standard code fragment search path, anywhere within the specified unit number range of the Unit Table. It invokes the driver's `Initialize` command, passing the `RegEntryIDPtr` to it. The driver's initialization code must return `noErr`

for `InstallDriverFromFragment` to complete successfully. This function returns the driver's `refNum` but it does not open the driver.

The unit table will grow to accept the new driver if the table has not already reached its maximum size; if `beginningUnit` is 48; and if `endingUnit` equals the highest unit number currently available, as returned by the `HighestUnitNumber` routine described on page 117.

RESULT CODES

`noErr`	0	No error
`badUnitErr`	–21	Bad unit number
`unitEmptyErr`	–22	Empty unit number
`unitTblFullErr`	–29	Unit table 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 that is in the Mac OS System folder, verifies that the file's contents are a native driver, and loads and installs the driver.

```
OSErr InstallDriverFromDisk
         (Ptr              driverName,
          RegEntryIDPtr    device,
          UnitNumber       beginningUnit,
          UnitNumber       endingUnit,
          DriverRefNum     *refNum);
```

`driverName`	Name of a disk file containing a driver
`device`	Pointer to entry in the Name Registry
`beginningUnit`	First Unit Table number of range acceptable for installation
`endingUnit`	Last Unit Table number of range acceptable for installation
`refNum`	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` to it. The driver's initialization code must return `noErr` for `InstallDriverFromDisk` to complete successfully. This function returns the driver's `refNum` but it does not open the driver.

The unit table will grow to accept the new driver if the table has not already reached its maximum size; if `beginningUnit` is 48; and if `endingUnit` equals the highest unit

number currently available, as returned by the `HighestUnitNumber` routine described on page 117.

RESULT CODES

`noErr`	0	No error
`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	retain 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 must 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,
                               refNum        *refNum);
```

`fragmentSpec`	pointer to a file system specification
`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`

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` to it. The driver's initialization code must return `noErr` for `InstallDriverFromFile` to complete successfully. This function returns the driver's `refNum` but it does not open the driver.

The unit table will grow to accept the new driver if the table has not already reached its maximum size; if `beginningUnit` is 48; and if `endingUnit` equals the highest unit number currently available, as returned by the `HighestUnitNumber` routine described on page 117.

RESULT CODES

`noErr`	0	No error
`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,
                       Str63            fragName,
                       RegEntryIDPtr    device,
                       UnitNumber       beginningUnit,
                       UnitNumber       endingUnit,
                       refNum           *refNum);
```

`memory`	pointer to beginning of fragment in memory
`length`	length of fragment in memory
`fragName`	An optional name of the fragment (primarily used 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`

DESCRIPTION

`InstallDriverFromMemory` installs a driver that is located in a CFM code fragment, using the standard code fragment search path, anywhere within the specified unit number range of the Unit Table. It invokes the driver's `Initialize` command, passing the `RegEntryIDPtr` to it. The driver's initialization code must return `noErr` for `InstallDriverFromMemory` to complete successfully. This function returns the driver's `refNum` but it does not open the driver.

The unit table will grow to accept the new driver if the table has not already reached its maximum size; if `beginningUnit` is 48; and if `endingUnit` equals the highest unit number currently available, as returned by the `HighestUnitNumber` routine described on page 117.

RESULT CODES

`noErr`	0	No error

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" beginning on page 102.

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" beginning on page 122.

```
OSErr InstallDriverForDevice
         (RegEntryIDPtr    device,
          UnitNumber       beginningUnit,
          UnitNumber       endingUnit,
          refNum           *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`

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` to it. The driver's initialization code must return `noErr` for `InstallDriverForDevice` to complete successfully. This function returns the driver's `refNum` but it does not open the driver.

The unit table will grow to accept the new driver if the table has not already reached its maximum size; if `beginningUnit` is 48; and if `endingUnit` equals the highest unit number currently available, as returned by the `HighestUnitNumber` routine described on page 117.

RESULT CODES

`noErr`	0	No error
`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 later. This service may be used by any software that loads or evaluates drivers.

```
short HigherDriverVersion (NumVersion *V1, NumVersion *V2);
```

```
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 non-released version*/
};
```

`V1` First version number being compared

`V2` Second version number being compared

DESCRIPTION

`HigherDriverVersion` returns 0 if `V1` and `V2` are equal. It returns a negative number if `V1` < `V2` and a positive number greater than 0 if `V1` > `V2`. If both drivers have `stage` values of `final`, a `nonRelRev` value of 0 is evaluated as greater than any nonzero number.

Driver Removal

Clients wishing to remove an installed driver should use `RemoveDriver`.

RemoveDriver

`RemoveDriver` removes an installed driver.

```
OSErr RemoveDriver   (refNum     refNum,
                      Boolean    Immediate);
```

`refNum` `refNum` of driver to remove

`Immediate` `true` means don't wait for driver to become idle

DESCRIPTION

`RemoveDriver` accepts a `refNum` 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`	`refNum` of driver to examine
`unit`	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
`driverLocation`	resulting CFM fragment locator (from which the 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 `unit`, 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 `driverLocation`, its CFM connection ID in `fragmentConnID`, its `DoDriverIO` entry point in `fragmentMain`, and its Driver Description in `driverDesc`.

Note
With 68K drivers, `GetDriverInformation` returns meaningful information in only the `unit`, `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`	`true`: return available units `false`: return allocated units
`returnedRefNums`	Maximum number of reference numbers to return; on completion, contains actual number of `refNums` returned
`refNums`	resulting array of returned `refNums`

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 `refNums` returned.

The sample code shown in Listing 7-8 uses `HighestUnitNumber` and `LookupDrivers` to print out the reference numbers of all installed drivers and obtain driver information.

RESULT CODES

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

Listing 7-8 Using the LookupDrivers function

```
FindAllDrivers ()
{
   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;

   GetDriverInformation (  aRefNum,
                           &theUnit,
                           &theFlags,
                           &theOpenCount,
                           theName,
                           &theDevice,
                           &theLoc,
                           &fragmentConnID,
                           &fragmentMain,
                           &theDriverDescription);
```

```
   printf ("Driver's flags are:  %x\n", theFlags);
}
```

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 section 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 runtime driver code in its expansion ROM. PCI devices without FCode and runtime driver code in ROM may not 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`

The OpenFirmware FCode in a PCI device's expansion ROM must provide and install a property with this name to have its driver appear in the Name Registry and be useful to the system during startup. 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, the following is a short description of the PCI boot sequence:

1. Hardware 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 persistant across system startups and are located in NVRAM are restored to their proper location in the Name Registry device tree.
6. If a PCI ROM device driver is marked as `kdriverIsLoadedUponDiscovery`, the driver is installed in the Device Manager Unit Table.
7. 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.
8. The Display Manager is initiated.
9. The SCSI Manager is initiated.
10. The Name Registry device tree is searched for PCI expansion ROM device drivers associated with device nodes.
11. The File Manager and Resource Manager are initialized.
12. Device properties that are persistant across system startups and located in the folder `System Folder:Preferences` are restored to their proper location in the Name Registry device tree.

At this point, PCI expansion ROM device drivers required for booting are loaded and initialized. What remains to be processed yet are device drivers that fall under Family Expert control. The steps that follow the above sequences are to allow for loading of disk-based experts and disk-based drivers. The steps are:

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-descriptor 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.
4. Scan the extensions folder for experts, (file type `'expt'`); load, initialize, and run the expert.
5. Run experts to scan the registry, using the driver-descriptor property associated with each node to determine which devices are of the appropriate family.
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. Unitialized 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
PCI 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 find 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 property string of type `compatible`. The comparison is always against the `nameInfoStr` in the DriverDescription 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 useable.
- When a device node has no driver, 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 zeroes, that use lower case for the letters a–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.

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. Break any ties using the driver's version number. (See "HigherDriverVersion" beginning on page 115).
3. If not installing the driver, return the driver at the head of the candidate list and go to step 7.
4. While we have candidates with which to attempt an installation, do the following:
5. Load and install the driver located at the head of the list.
6. The driver should probe the device, using DSL services, to verify the match. If the driver did not successfully initialize itself, go to step 4.
7. Discard any remaining candidates.

The routines that use this algorithm are described in detail in the sections that start with "Loading and Unloading" beginning on page 104.

▲ **WARNING**
You must try to match your driver with your device as securely as possible, using the routines and algorithms just described. If you fail to do so, the computer may crash with an unrecoverable bus error. ▲

IMPORTANT
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. ▲

The `DeviceProbe` function helps you determine if a device is present at an address.

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,`
	`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, i.e. if a bus error or a machine check is generated, it returns `noHardwareErr`.

RESULT CODES

`noErr`	0	Device present
`noHardwareErr`		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 device 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 looking for boot devices. Boot device nodes are flagged as `kdriverIsOpenedUponLoad` in the `DriverDescriptor` property associated with the device node. These devices are loaded, initalized, and opened by the system. These drivers must be in ROM and must be of service category `'ndrv'`. Drivers are passive code controlled by a client—in this case, the system starting up. PCI bridges are tagged `kDriverIsLoadedUponDiscovery` and `kDriverIsOpenedUponLoad`.

The second phase of startup comes after the Macintosh file system is available. In this second phase, the Extensions folder is scanned for Family Experts. Family Expert modules are developed by Apple and are not documented here. Family experts are run as they are located, and 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 DisplayManager 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 initalized and installed in the system in a family-specific way.

Note that startup steps 8 and 9 listed earlier in this section initiated 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 SCSI Interface Modules (SIMs) in the way described in *Inside Macintosh: Devices*. The Display Manager loads, initalizes 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" beginning on page xxiii.

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 an 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 device PCI 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" beginning on page 122. The `driverInfoStr` and `version` fields of the `DriverType` field of the two `DriverDescriptors` are compared, and the newer version of the driver is installed in the tree. When the driver is updated, the `DriverDescriptor` 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 (type `'DRVR'`). 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 `refNum`. 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" beginning on page 88.

```
OSErr ReplaceDriverWithFragment  (DriverRefNum theRefNum,
                                  ConnectionID 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" beginning on page 88.

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  theRefNum,
                      StringPtr     name);
```

`theRefNum`	Reference number of the driver to be renamed
`name`	Pointer to the driver's new name

DESCRIPTION

Given the Unit Table reference number of an installed driver in `theRefNum`, `RenameDriver` changes the driver's name to the contents of a string pointed to by `name`.

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-based programming interfaces and runtime architectures to their PCI native driver equivalents.

Resources of type `'DRVR'` in Mac OS are used to solve a broad variety of problems. Some `'DRVR'` resources drive devices through the use of an I/O Manager. 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 `'DRVR'` resources need not migrate to the new services and runtime. 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 not supplied for native drivers.

Device Manager Calls

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

In the AppleTalk implementation available with System 7.1 and earlier, 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. AppleTalk 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 12, "Network Drivers."

Power Manager

There are no Power Manager calls allowed from native device drivers.

Resource Manager

Resource Manager calls are not permitted, not even in the driver initalization routine. Instead, drivers use the Name Registry to manage initialization and configuration. The CFM provides dynamic loading of code fragments. See the discussion in "Driver Loader Library" beginning on page 102.

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 10, "Expansion Bus Manager."

Vertical Retrace Manager

Vertical blanking (VBL) facilites 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.

Gestalt Manager

Gestalt calls are available to 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.

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 will not provide any emulation facilities for device 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.

New Driver Services

This section describes new services that the Macintosh system software provides for native drivers.

Registry Services

Chapter 8, "Macintosh Name Registry," introduces the concept of a Name Registry. The Registry interface provides new mechanisms that allow drivers to expose information. Any data that might be of use to a configuration or debugging utility may be installed in the Registry and be directly available to the configuration application through the Registry programming interface.

▲ **WARNING**
Only a small set of Registry services are available at primary or secondary interrupt level. The set of services available at non-task level are gets and sets of properties associated with a single device entry. ▲

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
- `BlockCopy`

For a more complete set of driver services see Chapter 9, "Driver Services Library."

Timing Services

The Time Manager calls `InsTime`, `PrimeTime`, and `RmvTime` have been replaced with a new set of timing services. 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.

Note
The configuration and control sections of a native driver may use the Macintosh Toolbox. For further information, see "Separate Layers of Driver Implementation" beginning on page 51. ◆

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 9, "Driver Services Library."

Interrupt Mechanisms

To install and remove interrupt handlers you can use `InstallInterruptFunctions` and `RemoveInterruptFunctions`. These routines replace the earlier Slot Manager routines `SIntInstall` and `SIntRemove`.

The declarations for the interrupt handler, enabler, and disabler are the following:

```
typedef InterruptMemberNumber (*InterruptHandler)
                              (InterruptSetMember  member,
                               void *              refCon);

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. Primary, secondary and software-interrupt mechanisms are described in "Interrupt Management" beginning on page 190.

InstallInterruptFunctions

`InstallInterruptFunctions` installs interrupt control routines for an interrupt source.

```
OSStatus InstallInterruptFunctions
                     (InterruptSetID         set,
                      InterruptMemberNumber  member,
                      void *                 refCon,
                      Interrupt_Handler      handler,
                      Interrupt_Enabler      enableFunc,
                      Interrupt_Disabler     disableFunc;
```

`set`	IST interrupt set ID
`member`	Set member number
`refCon`	32-bit reference value
`handler`	Pointer to interrupt service routine (ISR)
`enableFunc`	Pointer to interrupt enabler routine (IER)
`disableFunc`	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, disabler, and acknowledge routines. Interrupt sets and the interrupt tree are discussed in "Interrupt Management" beginning on page 190.

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. A `nil` value may be passed in `handler`, `enableFunc`, or `disableFunc` to indicate no such control routine is available.

`InstallInterruptFunctions` returns `noErr` if the installation succeeded.

RESULT CODES

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

RemoveInterruptFunctions

`RemoveInterruptFunctions` removes interrupt control routines previously installed.

```
OSStatus RemoveInterruptFunctions
                                (InterruptSetID         set,
                                 InterruptMemberNumber  member);
```

`set`	IST interrupt set ID
`member`	Set member number

DESCRIPTION

`RemoveInterruptFunctions` removes the interrupt control routines from an interrupt source identified by the ID of an interrupt set in the interrupt tree and the number of a member in that set. Interrupt sets and the interrupt tree are discussed in "Interrupt Management" beginning on page 190.

RESULT CODES

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

Secondary Interrupt Services

The Deferred Task Manager call `DTInstall (dtTaskPtr: QElemPtr)` is replaced by `QueueSecondaryInterruptHandler` and `CallSecondaryInterruptHandler2`. These routines are discussed in "Secondary Interrupt Handlers" beginning on page 204.

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 part of the requesting application's context.

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. The Name Registry provides two kinds of persistant storage; see Chapter 8, "Macintosh 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 initalization 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" beginning on page 156, 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.

Macintosh 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 device driver and system software with a way 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 and may include such data as the physical location of the thing, technical descriptions of it, and logical addresses.

Names are created in the Name Registry by expert software. Each expert owns specific names and is responsible for removing them when they are no longer needed. Clients search for names 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 Macintosh 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 Macintosh Name Registry is less general; it is optimized for the specific needs of hardware and device driver configuration.

The Name Graph

Names in the Name Registry are connected together. At present the connections form a hierarchy, but in the future the names may be connected in a more general graph structure.

Note
Code must not depend on the order in which names are found in the Registry. ◆

Software finds new names in the Registry by locating names that it already knows and by examining names found nearby. By knowing what a name refers to, a program can find other names that might be used for a similar or related purpose.

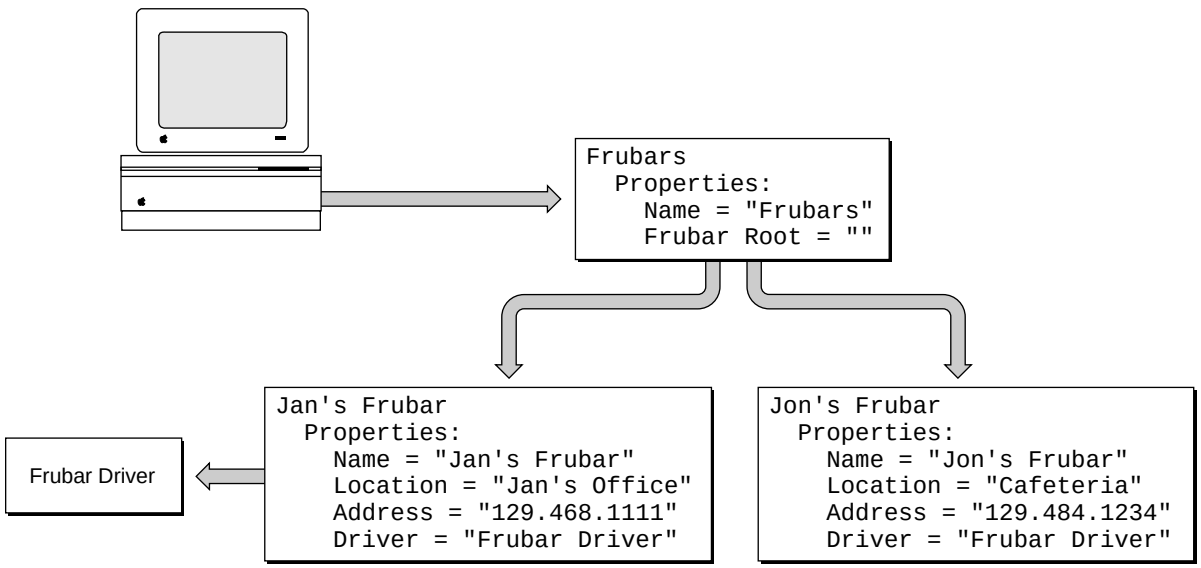
The hierarchical name graph is based on an origin name called the ***root.*** All names in the graph may be described by a pathway through the graph starting from the root. Future versions of the Registry may provide multiple paths to some names.

Name Properties

Each name 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, software can determine what the name identifies and what it can be used for.

Software uses Registry properties to find other software. For example, if a user specifies a name while running an application, the application may look the name up 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 8-1.

Figure 8-1 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 and a set of properties to the Registry for each device
3. find a driver for each device
4. initialize the driver

Connections between names are formed when names are added to the Registry. The connections have direction and point from an existing name to the new name.

The Expansion Bus Manager enters most of the names in the Registry by 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 as properties. Low-level experts are described in "Terminology" on page 47. In a few cases, drivers may enter names and properties in the Registry.

The software entity that creates a name owns the name, whether it is the Expansion Bus Manager, a low-level expert, or a device driver. Only the owner should remove a name. Since most device drivers do not enter names in the Registry, most drivers never remove them.

Name Registry Overview

This section summarizes the scope, design goals, limitations, and terminology of the Macintosh 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 names include the low-level experts and the Expansion Bus Manager. Clients include device drivers, control panels (resources of type `'CDEV'`), family experts, and other device-management software.

Limitations

The Name Registry supports a relatively small number of names. Other limitations include the following:

- Because all Registry contents reside in RAM, the number of names supported is limited by the available RAM space.
- Name creation and searching does not have to be fast.
- The Registry's information is volatile; it is lost when the system is restarted.

Terminology

The Macintosh Name Registry uses these special terms:

- *path:* a sequence of colon-separated names
- *name:* a null-terminated character string representing a thing or a concept

- *property:* a string and value pair associated with a name, which describes some characteristic of the thing represented by the name
- *modifier:* hardware- or implementation-specific information associated with a name or property. Modifier information is stored as bits in a 32-bit word

Using the Name Registry

This section describes the Macintosh Name Registry programming interface available to device drivers and other device control software in the second generation of Power Macintosh computers.

Coding Conventions

The header file declaring the Name Registry programming interface should be included as follows:

```
#include <NameRegistry.h>
```

No other header files are required.

Name Entry Management

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 path names. However, the Apple double colon (`::`) parent directory syntax is not currently supported.

Data Structures and Constants

Pathnames may be of any length, but components of a pathname are limited as follows:

```
enum
{
   kRegCStrMaxEntryNameLength    = 48
   kRegMaximumPropertyNameLength = 32
};

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 graph. The following constants indicate the direction of movement during traversals of the hierarchical Registry graph.

```
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
   kRegIterSubTrees  = 0x5L,  // include all subtrees of entry
   kRegIterSibling   = 0x6L,  // include all siblings
   kRegIterContinue  = 0x1L   // keep doing the same thing
};
```

Manipulating Entry Identifiers

Entry identifiers might contain allocated data, so Mac OS includes operations to copy and dispose of them.

RegistryEntryIDInit

`RegistryEntryIDInit` initializes a `RegEntryID` structure to a known, invalid state.

```
OSStatus RegistryEntryIDInit (RegEntryID *id);
```

id Identifier to be initialized

DESCRIPTION

Since `RegEntryID` values are allocated on the stack, it is not possible to detemine 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.

RESULT CODES

`noErr`	0	No error

RegistryEntryIDCompare

RegistryEntryIDCompare compares RegEntryID values to see if they are equal or if one is invalid.

```
Boolean RegistryEntryIDCompare ( const RegEntryID *id1,
                                 const RegEntryID *id2);
```

`id1`	First identifier
`id2`	Second identifier

DESCRIPTION

`RegistryEntryIDCompare` is useful for comparing two `RegEntryID` values to see whether they reference the same entry as well as to check if a `RegEntryID` contains 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` returns true when the other id is invalid. If both id values are `null`, the ids are consided equal and `RegistryEntryIDCompare` returns `true`.

RESULT CODES

`false`	0	id values different or id valid
`true`	1	id values equal or id invalid

RegistryEntryIDCopy

`RegistryEntryIDCopy` copies the identifier for a Name Registry entry, including its allocated data.

```
void RegistryEntryIDCopy   (const RegEntryID *src,
                            RegEntryID       *dst);
```

`src`	ID to be copied
`dst`	Destination ID

DESCRIPTION

Given the value of an existing `RegEntryID`, `RegistryEntryIDCopy` sets the value of another `RegEntryID` to be functionally the same.

RESULT CODES

`noErr`	0	No error

RegistryEntryIDDispose

`RegistryEntryIDDispose` disposes of a Name Registry identifier.

```
void RegistryEntryIDDispose (RegEntryID *id);
```

`id` `RegEntryID` value to be disposed of

DESCRIPTION

`RegistryEntryIDDispose` disposes of the identifier for a Name Registry entry pointed to by `id`, including its allocated data.

RESULT CODES

`noErr`	0	No error

Name Creation and Deletion

The following routines add new names and remove existing names from the Registry.

RegistryCStrEntryCreate

`RegistryCStrEntryCreate` creates a new child name entry in the Name Registry.

```
OSErr RegistryCStrEntryCreate (const RegEntryID     *parentEntry,
                               const RegCStrPathName*name,
                               RegEntryID           *newEntry);
```

`parentEntry`	`RegEntryID` value of the parent entry
`name`	name of the new entry as a C-string pathname
`newEntry`	returned `RegEntryID` value of the new entry

DESCRIPTION

Given the `RegEntryID` value of a Name Registry entry, `RegistryCStrEntryCreate` creates an entry that is its child, with the name `name`. It returns the `RegEntryID` value of the new entry.

If `parentEntry` is `NULL`, the name is assumed to be a rooted path. If the path begins with a colon, the path is relative to `parentEntry`. If the path does not begin with a colon, the path is a rooted path.

It is up to the caller to implement a current directory if it is needed.

RESULT CODES

`noErr`	0	No error

RegistryEntryDelete

`RegistryEntryDelete` deletes an entry from the Name Registry.

```
OSErr RegistryEntryDelete (const RegEntryIDPtr name);
```

`name` Pointer to `RegEntryID` of entry to delete

DESCRIPTION

Given the `RegEntryID` of an entry in the Name Registry, `RegistryEntryDelete` deletes it.

RESULT CODES

`noErr`	0	No error

Name Iteration and Searching

Writing code to traverse 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 iterations are provided, direction-oriented and search-oriented. The type of iteration is indicated by the call used to retrieve the next entry.

The Registry name iteration functions communicate through a `cookie` parameter with the following type:

```
typedef struct    RegEntryIter { void *opaque; }
                        RegEntryIter,
                        *RegEntryIterPtr;
```

RegistryEntryIterateCreate

The starting routine `RegistryEntryIterateCreate` starts an iteration over all device names.

```
OSErr RegistryEntryIterateCreate (RegEntryIter *cookie);
```

`cookie` Value used by all search routines

DESCRIPTION

`RegistryEntryIterateCreate` sets up the iteration process for finding device names in the Name Registry and returns a value in `cookie` that is used by `RegistryEntryIterate` or `RegistryEntrySearch`.

RESULT CODES

`noErr`	0	No error

RegistryEntryIterateSet

`RegistryEntryIterateSet` sets a `RegEntryIter` value to a specified starting entry and relationship.

```
OSStatus RegistryEntryIterateSet
                  (RegEntryIter        *cookie,
                   RegEntryIterationOp relationship,
                   const RegEntryID    *startEntryID);
```

DESCRIPTION

When a `RegEntryIter` structure is first created it is set to the root of the Name Registry with a relation of `kRegIterSubTrees`. `RegistryEntryIterateSet` lets you adjust

this starting point to a known entry and reset the relationship, so you can iterate or search over a subset of the device tree.

If `kRegIterContinue` is specified for `relationship`, the previous relationship used with the `RegEntryIter` structure is continued.

RESULT CODES

`noErr`	0	No error

RegistryEntryIterate

One kind of iteration call, `RegistryEntryIterate`, retrieves the next name in the Name Registry by moving in a specified direction.

```
OSErr RegistryEntryIterate
   (RegEntryIter           *cookie,
    RegEntryIterationOp    relationship,
    RegEntryID             *foundEntry,
    Boolean                *done);
```

`cookie`	Value used by all search routines
`relationship`	Search direction (values defined on page 140)
`foundEntry`	ID of the next entry found
`done`	`true` means searching is completed

DESCRIPTION

`RegistryEntryIterate` moves from name to name 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 of the next name found in `foundEntry`, or `true` in `done` if all names have been found.

RESULT CODES

`noErr`	0	No error

RegistryEntrySearch

Another kind of iteration call, `RegistryEntrySearch`, retrieves the next name in the Name Registry by moving in a specified direction and searching for a name with a matching property.

```
OSErr RegistryEntrySearch
   (RegEntryIter           *cookie,
    RegEntryIterationOp    relationship,
    RegEntryID             *foundEntry,
    Boolean                *done,
    const RegPropertyName  *propertyName,
    const void             *propertyValue,
    RegPropertyValueSize   propertySize);
```

`cookie`	Value used by all search routines
`relationship`	Search direction (values defined on page 140)
`foundEntry`	ID of the next entry found
`done`	`true` means searching is completed
`propertyName`	name of property to be matched
`propertyValue`	value of property to be matched
`propertySize`	size of property to be matched

DESCRIPTION

`RegistryEntrySearch` moves from name to name in the Name Registry, marking its position by changing the value of `cookie`. The direction of movement is indicated by `relationship`. `RegistryEntrySearch` returns the `RegEntryID` value of the next name found in `foundEntry`, or `true` in `done` if all names have been found.

`RegistryEntrySearch` returns only entries that match the values of `propertyName`, `propertyValue`, or `propertySize`. If the `propertyValue` pointer is `null`, then any property value matches.

RESULT CODES

`noErr`	0	No error

RegistryEntryIterateDispose

`RegistryEntryIterateDispose` disposes of the iteration structure after searching is finished.

```
void RegistryEntryIterateDispose (RegEntryIter *cookie);
```

`cookie`	Value used by all search routines

DESCRIPTION

Given the `cookie` value used previously, `RegistryEntryIterateDispose` disposes of resources used for searching.

RESULT CODES

`noErr`	0	No error

Name Lookup

`RegistryCStrEntryLookup` provides a fast forward search mechanism to find a name in the Registry quickly.

RegistryCStrEntryLookup

`RegistryCStrEntryLookup` finds an entry in the Name Registry quickly.

```
OSErr RegistryCStrEntryLookup
   (const RegEntryID       *searchPointID,
    const RegCStrPathName  *pathName,
    RegEntryID             *foundEntry);
```

`searchPointID`	`RegEntryID` value of starting point of search
`pathName`	Path name of entry to be found
`foundEntry`	`RegEntryID` value of found entry

DESCRIPTION

`RegistryCStrEntryLookup` finds an entry in the Registry based on `pathName`, starting from `searchPointID`.

If `searchPointID` is `null`, the path name is assumed to describe a rooted path. If the path name begins with a colon, the path is relative to `searchPointID`. If the path name does not begin with a colon, the path is a rooted path.

After using `RegistryCStrEntryLookup`, dispose of the `foundEntry` pointer by calling `RegistryEntryIDDispose`.

It is up to the caller to implement a current directory if it is needed.

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" beginning on page 143 should be used. ◆

RESULT CODES

noErr	0	No error

Path Name Parsing

The routines defined in this section convert a RegEntryID value to the equivalent full path name, give the path name's length, and parse the path name into its components.

RegistryEntryToPathSize

RegistryEntryToPathSize returns the size of the path name of a Registry entry.

```
OSErr RegistryEntryToPathSize
   (const RegEntryID    *entry,
    RegPathNameSize     *propertySize);
```

entry	RegEntryID value of a Name Registry entry
propertySize	Returned size in bytes of the path name to the entry

DESCRIPTION

RegistryEntryToPathSize returns in propertySize the length in bytes of the path name of the Name Registry entry designated by entry, including the path name's terminating character.

RESULT CODES

noErr	0	No error

RegistryCStrEntryToPath

RegistryCStrEntryToPath returns the path name and path name length of an entry in the Name Registry.

```
OSErr RegistryCStrEntryToPath
   (const RegEntryID    entry,
    RegCStrPathName     *propertyName,
    RegPathNameSize     propertySize);
```

entry	RegEntryID value of a Name Registry entry
propertyName	Returned path name to the entry
propertySize	Returned size in bytes of the path name to the entry

DESCRIPTION

Given the `RegEntryID` value of a Registry entry, `RegistryCStrEntryToPath` returns its path name in `propertyName` and the size in bytes of the path name, including the terminating character, in `propertySize`.

RESULT CODES

`noErr`	0	No error

RegistryCStrEntryToName

`RegistryCStrEntryToName` parses a Name Registry entry's path name, retrieves the last component of the path, and returns the ID of the entry's parent.

```
OSErr RegistryCStrEntryToName
   (const RegEntryID    *entry,
    RegEntryID          *parentEntry,
    RegCStrEntryName    *nameComponent,
    Boolean             *done);
```

`entry`	`RegEntryID` value of a Name Registry entry
`parentEntry`	Returned `RegEntryID` value of the entry's parent entry
`nameComponent`	Returned name of the entry as a C string
`done`	Returns `true` when done

DESCRIPTION

Given the `RegEntryID` value of a Registry entry, `RegistryCStrEntryToName` returns the `RegEntryID` value of its parent entry in `parentEntry` and the last component of its path name in `nameComponent`.

RESULT CODES

`noErr`	0	No error

Finding an Entry

Listing 8-1 is a sample of code that finds a device entry in the Name Registry.

Listing 8-1 Finding a name entry

```
void
TestFindDriverDeviceEntry(
        const char    *entryName
    )
{
    OSStatus         osStatus;
    RegEntryIter     cookie;
    Boolean          done;

    osStatus = RegistryEntryIterateCreate(&cookie);
    gInstanceCount = 0;                          /* How many did we find?  */
      if (osStatus == noErr) {
             done = FALSE;
             while (osStatus == noErr
                 && done == FALSE && gInstanceCount < kMaxInstanceCount) {
                   osStatus = RegistryEntrySearch(
                                &cookie,
                                kRegIterContinue,
                                &gDeviceEntryVector[gInstanceCount],
                                &done,
                                "name",
                                entryName,
                                strlen(entryName) + 1
                            );
             if (done == FALSE && osStatus == noErr) {
                 ++gInstanceCount;               /* We found an entry     */
             }
        }
        RegistryEntryIterateDispose(&cookie);
   }
}
```

The function shown in Listing 8-1 locates all entries named by the argument string. Note that it passes the length of the string + 1, because the Name Registry explicitly tests for the ending null byte. The function returns a vector of registry entry IDs and a count of the number it found.

Property Management

Properties describe what a name represents or how it may be used. Each name is associated with 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 (32 bytes). Typically, property names are ASCII strings, but null-terminated single-byte strings are also supported. Name property data structures and constants are listed in "Data Structures and Constants" on page 139.

Creation and Deletion

The routines described in this section add new properties or remove existing properties from an entry in the Name Registry.

RegistryPropertyCreate

`RegistryPropertyCreate` adds a new property to a Registry entry.

```
OSErr RegistryPropertyCreate
      (const RegEntryID          *entryID,
       const RegPropertyName     *propertyName,
       const void                *propertyValue,
       RegPropertyValueSize      propertySize);
```

`entryID`	`RegEntryID` value of a Name Registry entry
`propertyName`	Name of the property to be created
`propertyValue`	Value of the new property
`propertySize`	Size of the new property

DESCRIPTION

Given the `RegEntryID` value of a Registry 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`. Upon return, the value of `propertySize` will be the actual length of the value in bytes.

Property names may be any alphanumeric strings but may not contain slash (/) or semicolon (:) characters.

RESULT CODES

`noErr`	0	No error

RegistryPropertyDelete

`RegistryPropertyDelete` deletes a property from the Name Registry.

```
OSErr RegistryPropertyDelete
   (const RegEntryID          *entryID,
    const RegPropertyName     *propertyName);
```

`entryID` `RegEntryID` value of a Name Registry entry

`propertyName` Name of the property to be deleted

DESCRIPTION

`RegistryPropertyDelete` deletes the property named `entryID` from the Registry entry identified by `propertyName`.

RESULT CODES

`noErr`	0	No error

Property Iteration

Traversing the set of properties associated with a name is similar to iteration over names in the Registry, described in "Name Iteration and Searching" beginning on page 143.

Only one form of property iteration is provided—iteration over the set of properties associated with a single name.

A property iteration loop has this general form:

```
Create(...)
do {
      Iterate(...);
} while (!done);
Dispose(...);
```

Property iteration functions communicate by means of a cookie parameter that is a `RegPropertyIter` data structure:

```
typedef struct    RegPropertyIter { void *opaque; }
                        RegPropertyIter,
                        *RegPropertyIterPtr;
```

RegistryPropertyIterateCreate

The starting routine `RegistryPropertyIterateCreate` starts an iteration over all the properties associated with a name.

```
OSErr RegistryPropertyIterateCreate
   (const RegEntryID    *entry,
    RegPropertyIter     *cookie);
```

`entry`	`RegEntryID` value of a Name Registry name entry
`cookie`	Value used by all property search routines

DESCRIPTION

`RegistryPropertyIterateCreate` sets up the iteration process for finding the properties of the device name identified by `entry` and returns a value in `cookie` that is used by `RegistryPropertyIterate`.

RESULT CODES

`noErr`	0	No error

RegistryPropertyIterate

Repeated calls to `RegistryPropertyIterate` use the cookie returned by `RegistryPropertyIterateCreate` to find a succession of properties.

```
OSErr RegistryPropertyIterate
   (RegPropertyIter     *cookie,
    RegPropertyName     *foundProperty,
    Boolean             *done);
```

`cookie`	Value used by all property search routines
`foundProperty`	Name of the property found
`done`	`true` if all properties have been found

DESCRIPTION

`RegistryEntryIterate` moves from property to to property among the properties associated with the name specified in the prior `RegistryPropertyIterateCreate` call (see previous section). It returns the `name` of the next property found in `foundProperty`, or `true` in `done` if all properties have been found.

RESULT CODES

`noErr`	0	No error

RegistryPropertyIterateDispose

`RegistryPropertyIterateDispose` completes the property search process.

```
void RegistryPropertyIterateDispose (RegPropertyIter *cookie);
```

`cookie` Value used by all property search routines

DESCRIPTION

`RegistryPropertyIterateDispose` disposes of the resources 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

Property Retrieval and Assignment

The value of an existing property may be quickly retrieved or modified using the routines defined in this section.

RegistryPropertyGetSize

A property value may have any length. If the maximum length of a property is not known, use `RegistryPropertyGetSize` to determine its size so you can allocate space for its value.

```
OSErr RegistryPropertyGetSize
   (const RegEntryID          *entryID,
    const RegPropertyName     *propertyName,
    RegPropertyValueSize      *propertySize);
```

`entryID` `RegEntryID` value of a Name Registry entry

`propertyName` Name of the property

`propertySize` Returned size of the property

DESCRIPTION

`RegistryPropertyGetSize` returns in `propertySize` the length in bytes of the property named `propertyName` and associated with the name entry identified by `entryID`.

RESULT CODES

noErr	0	No error

RegistryPropertyGet

RegistryPropertyGet retrieves the value of a property in the Name Registry.

```
OSErr RegistryPropertyGet
   (const RegEntryID          *entryID,
    const RegPropertyName     *propertyName,
    void                      *propertyValue,
    RegPropertyValueSize      *propertySize);
```

entryID	RegEntryID value of a Name Registry entry
propertyName	Name of the property
propertyValue	Returned value of the property
propertySize	Size of the property

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.

RESULT CODES

noErr	0	No error

RegistryPropertySet

RegistryPropertySet sets the value of a property in the Name Registry.

```
OSErr RegistryPropertySet
   (const RegEntryID          *entryID,
    const RegPropertyName     *propertyName,
    const void                *propertyValue,
    RegPropertyValueSize      propertySize);
```

entryID	RegEntryID value of a Name Registry entry
propertyName	Name of the property

`propertyValue`	Value to set the property to
`propertySize`	Size of the property

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 buffer pointed to by `propertyValue`. Upon return, the value of `propertySize` will be the actual length of the new value in bytes.

RESULT CODES

`noErr`	0	No error

Standard Properties

Some standard Name Registry properties names are specified for device nodes. These names should not be used for other purposes. Standard reserved property names used by PCI expansion cards are listed in Table 8-1.

Table 8-1 Reserved Name Registry property names

Name	Description
Open Firmware standard properties	
`name`	name of the entry (`nameString`); see page 122
`reg`	the package's registers
`device_type`	the implemented interface
`interrupts`	defines the interrupts used
`model`	defines a manufacturer's model
`address`	defines large virtual address regions
`compatible`	defines alternate `name` property values
`status`	indicate the device's operations status
Properties defined by PCI Binding to Open Firmware	
`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
`alternate-reg`	alternate access paths for addressable regions
`assigned-addresses`	assigned physical addresses
`power-consumption`	function's power requirements
`vendor-id`	value from corresponding PCI configuration register

Table 8-1 Reserved Name Registry property names (continued)

Name	Description
`device-id`	value from corresponding PCI configuration register
`revision-id`	value from corresponding PCI configuration register
`class-code`	value from corresponding PCI configuration register
`min-grant`	value from corresponding PCI configuration register
`max-latency`	value from corresponding PCI configuration register
`devsel-speed`	value from corresponding PCI configuration register
`fast-back-to-back`	value from corresponding PCI configuration register
Properties specific to the Power Macintosh platform	
`fcode-rom-offset`	location of node's FCode in the expansion ROM
`driver-ref`	reference to driver controlling a specific entry
`driver-description`	property that contains the driver description structure
`driver-ist`	IST member and set value, used to install interrupt
`driver-ptr`	memory address of driver code
`AAPL,address`	vector to logical address pointers*
`AAPL,interrupts`	internal interrupt number
`AAPL,slot-name`	physical slot identifier
`height`	height in pixels (for display device node only)
`width`	width in pixels (for display device node only)
`linebytes`	number of bytes in each line (for display device node only)
`depth`	color depth of each pixel (for display device node only)
`driver,AAPL,MacOS, PowerPC`	driver code for Mac OS
`driver-reg,AAPL, MacOS,PowerPC`	descriptor for location of driver code for Mac OS

* See "Accessing Device Registers" beginning on page 235.

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 xxv. 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 8-1:

- 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 on request from AppleLink address APPLE.PCI. The `assigned-addresses` property tells the

driver where a card's relocatable address locations have been placed in physical memory.

- 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 (described in "Tools" beginning on page 343) can use the value of this property to determine the values of other properties, such as `driver-reg`. If a card's expansion ROM contains no FCode, the `fcode-rom-offset` property will be absent from the card's Name Registry node.
- 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-description` property is a structure taken from the driver header; it defines various characteristics of the device. For NuBus cards in a NuBus expansion chassis, a property of this type may be constructed from information in the slot resources of the card's expansion ROM. The contents of this property are defined in "Driver Description Structure" beginning on page 73.
- The `AAPL,address` property is a vector to an array of logical address pointers, as described in "Accessing Device Registers" beginning on page 235.
- The `AAPL,interrupts` property is an internal interrupt number that the Open Firmware startup process creates before any FCode is read from the card.
- 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. It's value may be different with different Power Macintosh models.
- The `height`, `width`, `linebytes`, and `depth` properties are attached to the nodes of graphic display devices to define the display's characteristics.
- The properties `driver,`*`xxx`*`,`*`yyy`*`,`*`zzz`* and `driver-reg,`*`xxx`*`,`*`yyy`*`,`*`zzz`* provide 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. It is usually better to use the `driver-reg` property, which is a descriptor for the location of the driver code associated with the device. 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.

Note
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 `driver` and `driver-reg` properties. 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 ◆

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 each namespace and are redefined for each namespace. The high-order 8 bits are reserved for each name and property set and are redefined for each name.

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 an modifier word:

Name	Value	Description
`kRegNoModifiers`	0x00000000	no entry modifiers in place
`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 names:

Name	Value	Description
`kRegEntryIsNameSpace`	0x00000001	a collection of like-named entries
`kRegEntryIsAlias`	0x00000002	read-only alias to another entry

The following constants have meaning for properties:

Name	Value	Description
`kRegPropertyValueIsSavedToNVRAM`	0x00000001	Saved in NVRAM
`kRegPropertyValueIsSavedToDisk`	0x00000002	Saved to disk

Modifier-Based Searching

Mac OS provides two iteration routines to simplify searching for names or properties that have particular modifiers. These routines are used with the search routines described in "Name Iteration and Searching" beginning on page 143.

RegistryEntryMod

`RegistryEntryMod` returns the next name that has a specified modifier.

```
OSErr RegistryEntryMod
   (RegEntryIter           *cookie,
    RegEntryIterationOp    relationship,
    RegEntryID             *foundEntry,
    Boolean                *done,
    RegEntryModifiers      matchingModifiers);
```

`cookie`	Value used by all search routines
`relationship`	Search direction (values defined on page 140)
`foundEntry`	ID of the next entry found
`done`	`true` means searching is completed
`matchingModifiers`	modifiers to be matched

DESCRIPTION

`RegistryEntryMod` moves from name to name in the Name Registry, marking its position by changing the value of `cookie`. The direction of movement is indicated by `relationship`. `RegistryEntryMod` returns the `RegEntryID` value of the next name found in `foundEntry`, or `true` in `done` if all names have been found.

`RegistryEntryMod` returns only name entries with modifiers that match the value of `matchingModifiers`.

RESULT CODES

`noErr`	0	No error

RegistryEntryPropertyMod

`RegistryEntryPropertyMod` returns the next name that has a property that has a specified modifier.

```
OSErr RegistryEntryPropertyMod
   (RegEntryIter           *cookie,
    RegEntryIterationOp    relationship,
    RegEntryID             *foundEntry,
    Boolean                *done,
    RegEntryModifiers      matchingModifiers);
```

`cookie`	Value used by all search routines
`relationship`	Search direction (values defined on page 140)
`foundEntry`	ID of the next entry found
`done`	`true` means searching is completed
`matchingModifiers`	modifiers to be matched

DESCRIPTION

`RegistryEntryPropertyMod` moves from name to name in the Name Registry, marking its position by changing the value of `cookie`. The direction of movement is indicated by `relationship`. `RegistryEntryPropertyMod` returns the `RegEntryID` value of the next name found in `foundEntry`, or `true` in `done` if all names have been found.

`RegistryEntryPropertyMod` returns only name entries with properties that have modifiers that match the value of `matchingModifiers`.

RESULT CODES

`noErr`	0	No error

Name Modifier Retrieval and Assignment

Existing names and parameters may have their modifier word's value set or retrieved. The caller is responsible for preserving bits that should remain set by reading the current value, modifying it, and then assigning the new value.

RegistryEntryGetMod

`RegistryEntryGetMod` fetches the modifiers for a name entry in the Registry.

```
OSErr RegistryEntryGetMod
   (const RegEntryID          *entry,
    RegEntryModifiers         *modifiers);
```

`entry` `RegEntryID` value of a Name Registry entry

`modifiers` Return value of modifiers

DESCRIPTION

`RegistryEntryGetMod` returns in `modifiers` the current modifiers for the name entry identified by `entry`.

RESULT CODES

`noErr`	0	No error

RegistryEntrySetMod

`RegistryEntrySetMod` sets the modifiers for a name entry in the Registry.

```
OSErr RegistryEntrySetMod
   (const RegEntryID          *entry,
    const RegEntryModifiers   modifiers);
```

`entry` `RegEntryID` value of a Name Registry entry

`modifiers` Value of modifiers to set

DESCRIPTION

`RegistryEntrySetMod` sets the modifiers specified in `modifiers` for the name entry identified by `entry`.

RESULT CODES

`noErr`	0	No error

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.

```
OSErr RegistryPropertyGetMod
   (const RegEntryID          *entry,
    const RegPropertyNamePtr  *name,
    RegPropertyModifiers      *modifiers);
```

`entry` `RegEntryID` value of a Name Registry entry

`name` Property name

`modifiers` Returned value of property modifiers

DESCRIPTION

`RegistryPropertyGetMod` returns in `modifiers` the current modifiers for the property with name `name` in the entry identified by `entry`.

RESULT CODES

`noErr`	0	No error

RegistryPropertySetMod

`RegistryPropertySetMod` sets the modifiers for a property in the Registry.

```
OSErr RegistryPropertySetMod
   (const RegEntryID          *entry,
    const RegPropertyNamePtr  *name,
    RegPropertyModifiers      modifiers);
```

`entry` `RegEntryID` value of a Name Registry entry

`name` Property name

`modifiers` Value of property modifiers to set

DESCRIPTION

`RegistryPropertySetMod` sets the modifiers specified in `modifiers` for the property with name `name` in the entry identified by `entry`.

RESULT CODES

`noErr`	0	No error

Code Samples

This section contains code samples that illustrate Name Registry operations.

Adding a Device

For all physical devices, adding a device to the 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 in the Registry and setting the appropriate parameter values. The example shown in Listing 8-2 adds a new name to the Registry with a single property.

Listing 8-2 Adding a name 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 node.
    */
   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 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 names in the Registry are connected to at least one other name, either an existing entry must be provided when creating a new entry or it will be assumed that the path is specified relative to the root.

Note
Although the current Registry supports only a hierarchy of names, future versions of the Registry may provide other kinds of connections between names. ◆

The creator of a name must determine where in the tree the name should appear. This determination may be made by convention, as shown in the foregoing example, or may be made by the user, running an administrative application.

Finding a Device

Every device driver typically needs to retrieve information about the device from the Registry. The example in Listing 8-3 retrieves the value of a single property for the specified name in the Registry.

Listing 8-3 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

When a device is to be permanently removed from the system, the information pertaining to the device must be removed from the Registry. When a name is removed from the Registry all properties associated with that name are automatically removed as well. Listing 8-4 illustrates removing a name from the Registry.

Note
In the current Macintosh system, all children of a parent node are removed when the parent is removed. Removing a parent node, thereby creating orphan nodes, may not be supported in future releases. ◆

Listing 8-4 Removing a name 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

Administrative software must be able to find various devices in the system. The example shown in Listing 8-5 contains two procedures. The first loops through names, invoking a callback function for each name. The second loops through the properties for a name, 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 nodes that match a complex set of search criteria.

Listing 8-5 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 kRegIterSubTrees.
    * 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;
}
```

Macintosh Name Registry

Driver Services Library

This chapter describes the routines that are provided for every native driver by the Macintosh Driver Services Library. The driver loader, part of Mac OS, automatically links the library to each generic driver when the driver is loaded. The routines included in the Driver Services Library implement all the system programming interfaces (SPIs) 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 Macintosh Toolbox. This chapter describes the services available in the device driver runtime 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 11 through 13.

Device Driver Execution Contexts

As explained in "Noninterrupt and Interrupt-Level Execution" beginning on page 54, 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. 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. Page faults are not allowed at this context.
- Noninterrupt level is the context where all other code is executed. Page faults are allowed at this context.

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" beginning on page 216. ◆

CurrentExecutionLevel

The function `CurrentExecutionLevel` lets code determine its execution context.

```
ExecutionLevel CurrentExecutionLevel (void);
```

DESCRIPTION

`CurrentExecutionLevel` returns one of the result codes shown below.

RESULT CODES

`kHardwareInterruptLevel`	Hardware interrupt level
`kSecondaryInterruptLevel`	Secondary interrupt level
`kTaskLevel`	Noninterrupt 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;
```

The symbol `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 void *KernelID;
```

You should use the derived types whenever possible to make their 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 just 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

This section presents a brief overview of the terminology and concepts used to describe native driver memory addressing.

An ***address space*** is the domain of addresses that can be directly referenced by the processor at any given moment. A logical address specifies a location within an address space. Logical addresses are always unsigned; the lower bound of a logical address is 0, and the upper bound is the size of the address space minus 1. For example, in a 4 GB address space there are 2^{32} (4 GB) distinct logical addresses for bytes, ranging from 0 to 2^{32}–1. The number of bits required to represent logical addresses (the size of the address) is often used to denote the size of the address space. For example, a 4 GB address space can also be called a 32-bit address space.

System 7 provides a single 32-bit address space that is in effect for all software. Future versions of Mac OS may provide distinct address spaces for different software entities.

I/O Operations and Memory

When I/O operations are performed between an external device and memory, several aspects of the memory's contents must be coordinated. Typically, the logical contents must be made physically resident so that they may be accessed at hardware interrupt or secondary interrupt level, where page faults are not allowed. Additionally, the coherency of data and instruction caches must be maintained. ***Memory coherency*** ensures that data being moved is not stale and that the effects of the data movement are always accessible to the processor.

When using DMA hardware to perform an I/O operation, it is also necessary to translate the logical address range into set of physical address ranges. This set of physical address ranges is called a ***scatter-gather list.***

Services are available which help a driver prepare a range of addresses for an I/O operation and clean up the range when the operation is finished. Routines in the Driver Services Library perform all cache manipulations, make the data physically resident, and generate a scatter-gather list. The client need not be concerned with the cache topology

or any other aspect of the hardware, because the Driver Services Library provides complete hardware isolation.

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 (direct memory access) I/O operations.

```
typedef void *PhysicalAddress;
```

Address spaces are referred to by values of type `AddressSpaceID`. The value `currentAddressSpaceID` refers to the current address space.

```
typedef KernelID AddressSpaceID;
enum
{
   kcurrentAddressSpaceID = 0
};
```

Memory Services Used During I/O Operations

Data transfers to and from memory require close cooperation with the memory system to ensure proper operation. The following considerations are particularly important:

- Physical memory must remain assigned to the I/O buffer for at least the duration of the transfer.
- Memory accesses may not generate page faults during logical I/O operations.
- Data transfers must maintain memory coherency. On output, it is essential that data in the processor's data cache be included in the transfer. On input, it is essential that the caches (both data and code) not be left with any out-of-date information
- It is desirable that the data cache contain at least some of the data that was transferred.
- Because cache architectures vary from processor to processor, it is desirable to minimize or eliminate processor dependencies in I/O drivers.

The I/O support services, when used properly, ensure that these considerations are met.

The most common I/O transaction is a single transfer where the I/O buffer belongs to the driver's client—for example, handling a page fault by reading data directly into the client's page.

Multiple transactions may also occur with a single buffer. An example is a network driver whose transactions consist of reading data into its own buffer, processing the data, then copying the data to a client's buffer. In this case, the driver reuses the same buffer for an indefinite number of transactions.

The two I/O transaction services that the Driver Services Library provides are `PrepareMemoryForIO` and `CheckpointIO`. `PrepareMemoryForIO` tells the operating system that a particular buffer will be used for I/O transfers. It assigns physical memory to the buffer and, optionally, prepares the processor's caches for a transfer. `CheckpointIO` tells the operating system that the previously started transfer, if any, is complete. It also specifies whether there will be more transfers and may specify the direction of the next transfer. It finalizes the caches and, if the next I/O direction is specified, prepares the caches for that transfer. If its parameters specify that no more transfers will be made, `CheckpointIO` deallocates the resources associated with the buffer preparation: subsequent I/O operations on this range of memory must begin again with a call to `PrepareMemoryForIO`.

A single transfer uses a `PrepareMemoryForIO` call before the transfer and a `CheckpointIO` call following. The `PrepareMemoryForIO` parameters specify the buffer location and the I/O direction, while the `CheckpointIO` parameters specify that no more transfers will be made.

Multiple transfers use a `PrepareMemoryForIO` call when the buffer is allocated, a `CheckpointIO` call before each transfer, and a `CheckpointIO` call when the buffer is deallocated. The `PrepareMemoryForIO` parameters specify the buffer location, but might or might not specify the I/O direction. The I/O direction is omitted if the transfer is not imminent, because the cache preparation would be wasted. The `CheckpointIO` calls before each transfer specify the direction of the transfer and specify that more transfers will be made. A `CheckpointIO` call is not needed before the first transfer, if the `PrepareMemoryForIO` parameters specified an I/O direction. The final `CheckpointIO` parameters specify that no more transfers will be made.

▲ **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. A native driver can call `PrepareMemoryForIO` from its `doDriverIO` handler. The `doDriverIO` entry point is discussed beginning on page 78.

PrepareMemoryForIO Data Structures

The `PrepareMemoryForIO` function, described in "Using PrepareMemoryForIO" beginning on page 184, has a single parameter, a pointer to an `IOPreparationTable`

structure. The last fields of the `IOPreparationTable` structure contain pointers to three other structures:

- A `LogicalMappingTable` structure, which contains a list of logical addresses of allocated areas in memory
- A `PhysicalMappingTable` structure, which contains the physical addresses of the same areas
- An `AddressRangeTable` structure, which contains the corresponding pairs of base addresses and range length values

These structures are declared 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;
};

struct LogicalAddressRange {
   LogicalAddress    address;
   ByteCount         count;
};

typedef struct LogicalAddressRange *LogicalAddressRangePtr;
typedef LogicalAddress     *LogicalMappingTablePtr;

struct PhysicalAddressRange {
   PhysicalAddress   address;
   ByteCount         count;
};

typedef struct PhysicalAddressRange *PhysicalAddressRangePtr;
typedef PhysicalAddress    *PhysicalMappingTablePtr;
```

```
struct AddressRange
{
   void        *base;
   ByteCount   length;
};

typedef struct AddressRange   AddressRange;
typedef struct AddressRange   *AddressRangeTablePtr;

struct MultipleAddressRange
{
   ItemCount            entryCount;
   AddressRangeTablePtr rangeTable;
};
```

Fields of the `IOPreparationTable` structure may contain these values:

```
typedef OptionBits IOPreparationOptions;
enum {
   kIOMultipleRanges          = 0x00000001,
   kIOLogicalRanges           = 0x00000002,
   kIOMinimalLogicalMapping   = 0x00000004,
   kIOShareMappingTables      = 0x00000008,
   kIOIsInput                 = 0x00000010,
   kIOIsOutput                = 0x00000020,
   kIOCoherentDataPath        = 0x00000040,
   kIOClientIsUserMode        = 0x00000080
};

typedef OptionBits IOPreparationState;
enum {
   kIOStateDone   = 0x00000001
};
```

The `IOPreparationTable` structure specifies the memory ranges to be prepared for I/O use and provides storage for the mapping information that is returned.

Fields of the `IOPreparationTable` structure contain the following information:

- The `options` field specifies optional characteristics of the `IOPreparationTable` structure and the transfer process. Possible values in this field are discussed in "Mapping Table Options" beginning on page 183.
- The `state` field is filled in by `PrepareMemoryForIO` to indicate the state of the `IOPreparationTable` structure. `IOPreparationTableState` may contain `kIOStateDone` to indicate that the `PrepareMemoryForIO` function successfully prepared memory up to the end of the specified ranges. Regardless of whether or not

state is set to kIOStateDone, the firstPrepared and lengthPrepared fields of IOPreparationTable indicate the range prepared.

- The preparationID field is filled in by PrepareMemoryForIO to indicate the identifier that represents the I/O transaction. When the I/O operation has been completed or abandoned,the IOPreparationID value is used to finish the transaction, as described in "Finalizing I/O Transactions" beginning on page 185.
- The addressSpace field specifies the address space containing the logical ranges prepared. If the address space is specified by the kCurrentAddressSpaceID value, PrepareMemoryForIO changes it to the actual AddressSpaceID value unless the ranges are in global areas. With Mac OS 7.5, there is only one address space, which must be specified as kCurrentAddressSpaceID.
- The granularity field helps PrepareMemoryForIO deal with partial preparation of memory. It is useful for transfers with devices that operate on fixed-length buffers. The lengthPrepared value becomes 0 or an integral multiple of the value of granularity rounded up to the next greatest page alignment, preventing the preparation of more memory than the caller needs. A value of 0 for granularity specifies no granularity. No check is made to determine whether the specified range lengths are multiples of granularity.
- The firstPrepared field specifies the byte offset to the range at which to begin preparation. Note that when a MultipleAddressRange is specified, this offset points to the aggregate range.
- The lengthPrepared field is filled in by PrepareMemoryForIO to indicate the number of bytes, starting at firstPrepared, that were prepared. This is true regardless of whether the preparation was partial or complete. The firstPrepared and lengthPrepared fields control partial preparations. When calling PrepareMemoryForIO the first time, specify 0 for firstPrepared. If the tableState returned does not indicate kIOStateDone, a partial preparation was performed. After the transfer and final CheckPointIO are made against this preparation, another PrepareMemoryForIO call can be made to prepare as much as possible of the ranges that remain. This time, specify firstPrepared as the sum of the current firstPrepared value and lengthPrepared. Prepare, transfer, and final checkpoint actions can be repeated in sequence until IOPreparationState contains kIOStateDone.
- The mappingEntryCount field specifies the number of entries in the logicalMapping or physicalMapping structures. The logicalMapping structure is assumed to have two entries per range if kIOMinimalLogicalMapping is specified, regardless of mappingEntryCount. One entry per page is needed. If there are not enough entries, PrepareMemoryForIO performs a partial preparation within the limit of the tables and the kIOStateDone state bit returns 0.
- The logicalMapping field gives the address of the LogicalMappingTable structure filled in by PrepareMemoryForIO. This structure is optional; a nil value in logicalMapping specifies that there is no table. The table needs to have as many entries as there are distinct logical pages in the ranges. On return from PrepareMemoryForIO, LogicalMappingTable contains the static logical addresses corresponding to the physical addresses of the ranges. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings is the exact static logical mapping of the first prepared address

in that range, regardless of page alignment, while the remaining entries are page-aligned. If `kIOMinimalLogicalMapping` is in the `options` field, `PrepareMemoryForIO` returns only the first and last static logical mappings of each range.

- The `physicalMapping` field specifies the address of the `PhysicalMappingTable` structure to be filled in by `PrepareMemoryForIO`. The table needs to have as many entries as there are distinct logical pages in the range. This structure is optional; a `nil` value specifies that there is no table. On return, the `PhysicalMappingTable` structure contains the physical addresses that specify the ranges. The table is a concatenation, in order, of the mappings for each specified range. The first entry of each range's mappings is the exact physical mapping of the first prepared address in that range, regardless of page alignment, while the remaining entries are page-aligned.
- The `rangeInfo` field specifies the ranges to prepare. If `kIOMultipleRanges` is omitted from the `options` field, `rangeInfo` is interpreted as an `AddressRange` structure with the name `'range'`. If `kIOMultipleRanges` is specified, `rangeInfo` is interpreted as a `MultipleAddressRange` structure with the name `'multipleRanges'`. The `firstPrepared` specification determines the range table entry in which to begin preparation. Prior ranges, if any, will not be prepared and need not have mapping table space allocated for them.
- `AddressRange` fields have the following meanings.
  - The `base` field specifies the lowest address in the range. If the `kIOLogicalRanges` option is omitted, `base` is treated as a physical address. If `kIOLogicalRanges` is specified, `base` is treated as a logical address in the address space specified by the `addressSpace` field.
  - The `length` field specifies the length of the range.
- `MultipleAddressRange` fields specify an array of `AddressRanges`, such as in a scatter-gather specification. `MultipleAddressRange` fields have the following meanings.
  - The `entryCount` field specifies the number of entries in `rangeTable`.
  - The `rangeTable` field specifies the address of an array of `AddressRange` structures, as described earlier. It is acceptable for the specified ranges to overlap, either directly by specification or indirectly by being located on the same pages.

When using the structures just described, observe these cautions:

- `PrepareMemoryForIO` guarantees that the underlying physical memory remains assigned to the ranges until `CheckpointIO` relinquishes it. However, it does not guarantee that the original logical address ranges remain mapped. In particular, the controlling areas may be deleted before `CheckpointIO` is called. If the caller cannot guarantee that the areas will continue to exist, logical address references to the underlying physical memory must be made through the static logical addresses provided in the mapping tables. This is not a problem with Mac OS 7.5 because Mac OS 7.5 does not provide the equivalent of area deletion. However, it is advisable to use the logical mapping information rather than the original range specification anyway, for compatibility with future Mac OS releases.
- There are no explicit length fields in the mapping tables. Instead, entry lengths are implied by the entry's position in the table, the overall range length, and the page size. In the general case, the length of the first entry is to the next page alignment, the

lengths of the intermediate entries are the page size, and the length of the last element 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 `preparedLength`.

- Recognize the resource and execution time consequences of supplying partial mapping information. Supplying neither a full physical or full logical mapping table makes the system allocate at least one full table. Sharing one full table with the system, but not both full tables, results in computational overhead when `CheckpointIO` is called more than once.

Mapping Table Options

This `options` field of the `IOPreparationTable` structure contains bit subfields 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. With Mac OS 7.5, physical address range specification is not supported and `kIOLogicalRanges` must be used.
- `kIOMinimalLogicalMapping` specifies that the `LogicalMappingTable` structure is to be filled in with just the first and last mappings of each range, arranged in pairs. This mapping is useful for transfers where physical addresses are used for the bulk of the transfer, but logical addresses must be used to handle unaligned portions at the beginning and end—it makes it unnecessary to allocate a full-sized logical mapping table. Two entries per range are used, regardless of the range sizes. However, the value of the second entry of a pair is undefined if the pair's range is contained within a single page.
- `ioShareMappingTables` specifies that the system can use the caller's mapping tables instead of maintaining its own copies of the tables. This option conserves system resources. It can be specified only if the mapping tables are located in logical memory that can not generate a page fault (such as a resident area or a locked portion of a pageable area), and that will remain so until the final `CheckpointIO` process finishes. The mapping tables must also remain allocated and the entries unaltered until after the final `CheckpointIO`. It is not necessary for the caller to provide both tables.
- `kIOIsInput` specifies that data will be moved into main memory.
- `kIOIsOutput` specifies that data will be moved out of main memory. The values of `kIOIsInput` and `kIOIsOutput` are independent. You may specify either, both, or neither at preparation time. Specifying neither is useful when the preparation must be made long in advance of the transfer, so that system resources are allocated. `CheckPointIO` can be then be called just prior to the transfer to prepare the caches.
- `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. When in doubt, omit this option. Incorrectly omitting this option merely slows the operation of the computer, whereas incorrectly specifying this option can result in erroneous behavior and crashes.

- `kIOClientIsUserMode` indicates that `PrepareMemoryForIO` is being called on behalf of a non-privileged 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 Mac OS 7.5. 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 with the system, the main processor caches and other data transfers. 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 non-cacheable.

```
OSStatus PrepareMemoryForIO   (IOPreparationTable*
                               theIOPreparationTable);
```

`PrepareMemoryForIO` returns these error codes:

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

The driver or other software must perform I/O preparation before permitting data movement. For operations with block-oriented devices, the preparation is best done just before moving the data, typically by the driver. For operations upon buffers (such as memory sharing between the main processor and a coprocessor, frame buffer, or buffer internal to a driver) the preparation is best performed when the buffer is allocated.

If insufficient resources are available to prepare all of the specified memory, `PrepareMemoryForIO` will prepare as much as possible, and indicate to the caller which memory ranges were prepared. This is called *partial preparation.* The caller can examine the `kIOStateDone` bit in the `state` field of the `IOPreparationTable` structure to check whether the preparation was complete or partial. The caller can determine in either case which part of the overall range was prepared by examining the `firstPrepared` and `lengthPrepared` fields.

Memory must be prepared and finalized for the benefit of the system and other users of the memory and backing store, even if the caller does not need any of the information provided by `PrepareMemoryForIO`.

Calls to `PrepareMemoryForIO` should be matched with calls to `CheckpointIO`, even if the I/O operation was abandoned. In addition to applying finishing operations to the memory range, `CheckpointIO` deallocates system resources used in preparing the range, as discussed in the next section.

Finalizing I/O Transactions

This section describes the `CheckpointIO` function and its options.

```
typedef OptionBits IOPreparationOptions;
enum{
   knextIsInput   = 0x00000001,
   knextIsOutput  = 0x00000002,
   kmoreTransfers = 0x00000004
};

OSStatus CheckpointIO (IOPreparationID       PreparationID,
                       IOPreparationOptions  Options);
```

`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 is abandoned.

`CheckpointIO` returns these error codes:

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

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.

`PreparationID` is the `IOPreparationID` made for the I/O operation, as returned by a previous call to `PrepareMemoryForIO`. This ID is not valid following `CheckpointIO` if the `moreTransfers` option is omitted.

`Options` specifies optional operations. Values for this field are the following:

- `kNextIOIsInput` specifies that data will be moved into main memory
- `kNextIOIsOutput` specifies that data will be moved out of main memory

`kMoreIOTransfers` specifies that further I/O transfers will occur to or from the buffer. This option is required if either `kNextIOIsInput` nor `kNextIOIsOutput` is specified. It is also useful if the caller knows there will be another transfer but cannot specify the direction. If `moreTransfers` is omitted, all system resources associated with memory preparation are reclaimed, including allocated data structures and the value of `IOPreparationID` .

Note
The values of `kNextIOIsInput` and `kNextIOIsOutput` are independent. You may specify either, both, or neither. Specifying neither is useful for finalizing the previous transfer when the next transfer is not immediately pending. ◆

Cache Operations

The Driver Services Library provides several routines and data types that drivers can use for cache memory operations.

```
ByteCount GetLogicalPageSize (void);
ByteCount GetDataCacheLineSize (void);

OSStatus FlushProcessorCache     (AddressSpaceID   spaceID,
                                  LogicalAddress   base,
                                  ByteCount        length);

typedef unsigned longProcessorCacheMode;
enum {
   kProcessorCacheModeDefault       = 0,
   kProcessorCacheModeInhibited     = 1,
   kProcessorCacheModeWriteThrough  = 2,
   kProcessorCacheModeCopyBack      = 3
};

typedef unsigned long PageStateInformation;
enum {
   kPageIsProtected           = 0x00000001,
   kPageIsProtectedPrivileged = 0x00000002,
   kPageIsModified            = 0x00000004,
   kPageIsReferenced          = 0x00000008,
   kPageIsLocked              = 0x00000010,
   kPageIsResident            = 0x00000020,
   kPageIsShared              = 0x00000040,
   kPageIsWriteThroughCached  = 0x00000080,
   kPageIsCopyBackCached      = 0x00000100
};

struct PageInformation
{
   AreaID               area;
   ItemCount            count;
   PageStateInformation information [1];
};

typedef struct PageInformation PageInformation,
               *PageInformationPtr;
```

```
OSStatus GetPageInformation   (AddressSpaceID   addressSpace,
                               LogicalAddress   base,
                               ByteCount        length,
                               PBVersion        version,
                               PageInformation  *pageInfo);

OSStatus SetProcessorCacheMode(AddressSpaceID      addressSpace,
                               LogicalAddress      base,
                               ByteCount           length,
                               ProcessorCacheMode  mode);
```

`FlushProcessorCache`, `GetPageInformation`, and `SetProcessorCacheMode` return these error codes:

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

`GetPageInformation` 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 `PageInformation` to be returned, thereby providing backward compatibility. The `pageInformationVersion` parameter is the version of `PageInformation` defined in the current interface.

Parameter `thePageInfo` is filled in with information about each logical page. This buffer must be large enough to contain information about the entire range. Page information is as follows:

- `theCount` indicates the number of enties in which information was returned.
- `theInformation` contains one `PageStateInformation` entry for each logical page.

The bits of `PageStateInformation` are as follows:

- `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 non-pageable) 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.

The `FlushProcessorCache` function forces data from cache out to main memory. The `SetProcessorCacheMode` function forces the cache mode for selected pages of memory. These functions lets special-purpose drivers optimize their I/O performance.

▲ **WARNING**
Take care when using the `FlushProcessorCache` and `SetProcessorCacheMode` functions, because they may conflict with the cache mode operations of Mac OS. Most drivers need use only `PrepareMemoryForIO` and `CheckPointIO`. ▲

```
void SynchronizeIO (void)
```

The `SynchronizeIO` routine executes the PowerPC EIEIO (Enforce In-order Execution of I/O) instruction. This ensures in-order code execution between accesses to noncached devices.

Memory Allocation and Copying

The Driver Services Library provides services to allocate and free system memory for device drivers. You should always use these services to obtain dynamic memory.

Memory allocations can be performed only at noninterrupt execution level. Memory deallocations can be performed at either noninterrupt or software interrupt level. Execution levels are discussed in "Driver Execution Contexts" beginning on page 71.

Allocating Resident Memory

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 `GrowProc` was called and the pool is exhausted.

```
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.

`PoolAllocateResident` returns these error codes:

`noErr`	0	No error
`memFullErr`		
`qErr`		

Allocating a Buffer

`MemAllocatePhysicallyContiguous` allocates a buffer that is resident and is guaranteed to be physically continuous. It returns the buffer's logical address.

```
LogicalAddress MemAllocatePhysicallyContiguous
         (ByteCount  theLength,
          Boolean    clear);
```

`theLength` The number of bytes of memory to allocate.

`clear` Whether or not the allocated memory is to be zeroed.

`MemAllocatePhysicallyContiguous` returns these error codes:

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

Driver code can pass the address returned by `MemAllocatePhysicallyContiguous` to `PrepareMemoryForIO` (described on page 184) to obtain the buffer's physical location.

Deallocating Memory

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`. Similarly, `MemDeallocatePhysicallyContiguous` deallocates memory allocated by `MemAllocatePhysicallyContiguous`.

```
OSStatus PoolDeallocate (void *Address);
```

`Address` Address of pool memory chunk to deallocate.

`PoolDeallocate` returns these error codes:

`noErr`	0	No error
`memFullErr`		
`qErr`		

```
OSStatus
MemDeallocatePhysicallyContiguous (void *theBlockAddress);
```

`theBlockAddress` Address of the memory block to free.

`MemDeallocatePhysicallyContiguous` returns these error codes:

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

Copying Memory To Memory

The `BlockCopy` routine copies the chunk of memory at `srcPtr` to `destPtr`. Parameter `byteCount` specifies how many bytes are copied. `BlockCopy` uses the most appropriate version of `BlockMove`, as described in "BlockMove Routines" beginning on page 234.

```
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

Interrupt Management

This section discusses interrupt management for native drivers in the second generation of Power Macintosh computers. A general description of the new interrupt model is given first, followed by a detailed description of its programming interface.

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 ***primary interrupt handler*** and a ***secondary interrupt handler.*** The primary 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 primary 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" beginning on page 54.

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 services 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, a new native driver interrupt model is introduced that replaces the traditional interrupt-handling mechanisms used in previous Macintosh computers. This new model provides a more standardized execution environment for interrupt processing by using two key strategies:

- The new model formalizes the concept of primary and secondary interrupt levels for processing interrupts. Primary interrupt level execution happens as a direct result of a hardware interrupt request. Secondary interrupt level provides a way to defer non-critical 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 PCI card or device is represented by a node in hierarchical tree, called an ***Interrupt Source Tree (IST).*** Generally the leaf nodes of the tree represent interrupt sources for devices and the parent nodes representing 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 needs 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 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 more detailed description of these concepts follows.

Primary and Secondary Interrupt Levels

Primary interrupt level is also called *hardware interrupt level.* Primary interrupt level execution happens as a direct result of a hardware interrupt request. To insure maximum system performance, primary 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. Primary 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 primary interrupt handlers are listed in "***Services That Can Be Called From the Primary Interrupt Level***" on page 217.

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 primary 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 primary 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 "***Services That Can Be Called From Secondary Interrupt Handlers***" on page 217.

Note
The execution of secondary interrupt handlers may be interrupted by primary interrupts. ◆

When writing device drivers that handle hardware interrupts, it is important to balance the amount of processing done within the primary 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 primary interrupt level to secondary interrupt level and from secondary interrupt level to the driver's main task. 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

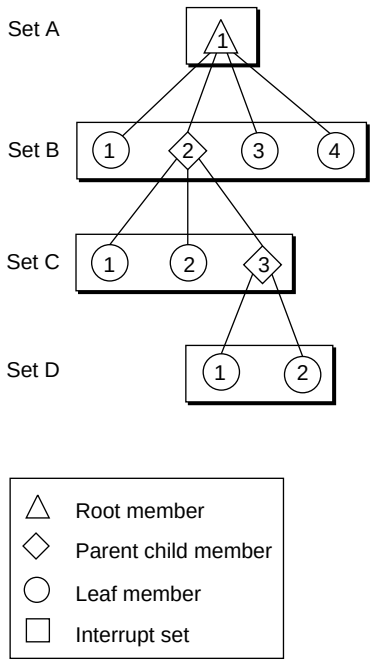
An Interrupt Source Tree is composed of hierarchically arranged nodes. Each node represents a distinct hardware interrupt source. Nodes are called *interrupt members* and are arranged in *interrupt sets.*

An ***interrupt set*** is identified by an `InterruptSetID` 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 a `InterruptSetID` and `InterruptMemberNumber` 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 9-1 illustrates a simplified example of an Interrupt Source Tree.

Figure 9-1 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 a interrupt service routine (ISR)
- a pointer to a interrupt enabler routine (IER)
- a pointer to a 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 route the processing of an interrupt.

When a handler ISR is invoked, it is supplied with three parameters. The first two parameters indicate the source of the interrupt with an `InterruptSetID` and `InterruptMemberNumber`, forming an `InterruptSetMember`. This allows a single ISR registered with multiple interrupt sources to determine which of the multiple

sources is the current interrupt. The third parameter is the reference constant value that was registered along with the ISR. The reference constant is not used by the system, and its use is completely up to the driver writer.

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 a 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 root member's child members should be designated as the current parent member for continued categorization of the interrupt and returns the `InterruptMemberNumber` 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 the leaf member's ISR is `null`, the interrupt request is dismissed as a spurious interrupt and ignored.

Consider an example using the simplified IST diagrammed in Figure 9-1 on page 193. 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 services the requesting device and returns `kIsrIsComplete`.

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 towards leaf members gives members closer to the root of the tree a higher priority. Hence, the hierarchichal 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 subsequently invoked. This could be an antistarvation algorithm or a priority based on the value of `InterruptMemberNumber`. What ever 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.

Interrupt Source Tree Construction

The Open Firmware startup process performs the initial construction and maintenance of the system's IST for all built-in I/O and single-function PCI expansion cards.

Note
Expansion card developers normally have no need to construct the IST, but may need to extend it as described in "Extending the Interrupt Source Tree" beginning on page 196. The following description of the initial construction process is included for completeness. ◆

As each interrupt member in the IST is created, a null ISR is installed and the IER and IDR routines are inherited from the parent member. Subsequent software can then replace the default routines by installing ones that are tailored to the newly-created interrupt member's interrupt source.

As the IST acquires each new interrupt set, the parent member has a transversal ISR installed on top of the null ISR the parent member acquired when it was created as part of a previous interrupt set. The interrupt members in the new interrupt set inherit an IER and IDR from the parent member. If the 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. Otherwise, each parent member's IER or IDR moderates interrupts transparently to the caller of a member's IER or IDR.

The construction process is repeated until the entire IST is in place.

IMPORTANT
Default enablers, disablers and transversal ISRs for all Macintosh built-in I/O devices are provided and installed by Apple I/O experts. Drivers that use them 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 8. As explained in "Initialization and Finalization Routines" beginning on page 83, a driver's initialization command call

contains a `RegEntryID` value that refers to the set of 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 8-1 on page 156. 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, 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

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 single-function PCI expansion cards.

However, PCI cards that extend the base system do not automatically receive nodes in the IST. Examples of the types of devices that extend the system are multi-function cards and PCI-to-PCI expansion bus cards. Because these cards 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 nor the IST. ◆

If you are extending the system by means of PCI bus slots or a multi-function 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 multi-function) 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 Runtime Structure" beginning on page 76.
- Use the `GetInterruptFunctions` function with the slot's `InterruptSetMember` 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` 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 multi-function 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" beginning on page 142.
- 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 8-1 on page 156.
- 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 resister 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" beginning on page 150. 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.

IMPORTANT
There is no removal mechanism for sets or members. The current release of Mac OS does not yet support hot plug-and-play devices. ▲

Basic Data Types

This section defines some data types and values that are fundamental to interrupt management.

```
typedef KernelID  InterruptSetID;
typedef long      InterruptMemberNumber;

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 Boolean InterruptSourceState;
enum
{
   kSourceWasEnabled    = true,
   kSourceWasDisabled   = false
};
```

Interrupt Return Values

Interrupt handlers return these values to the system:

```
typedef long InterruptReturnValue;
enum
{
   kFirstMemberNumber   =  1,
   kMemberNumberParent  = -2,
   kIsrIsNotComplete    = -1,
   kIsrIsComplete       =  0
};
```

If a child node in the Interrupt Tree returns `kIsrIsNotComplete`, the system automatically invokes its siblings and even its ancestors. It sets the dispatcher to polling handlers looking for one to handle the interrupt. A handler that successfully handles the interrupt returns `kIsrIsComplete`.

Control Routines

This section describes several interrupt control routines.

```
InterruptMemberNumber InterruptHandler
                        (InterruptSetMember  member,
                         void *              refCon);
```

When an ISR is invoked, `member` contains the member set ID and set member number of the currently interrupting source. This allows a single ISR to handle multiple interrupt members. `RefCon` contains the reference constant that was installed along with the ISR.

If the ISR returns `kIsrIsComplete`, the interrupt dispatcher stops any further traversal of the Interrupt Service Tree and treats the interrupt request as serviced. If the ISR returns a positive number, the dispatcher uses that number to identify which child member should be invoked next. If `kMemberNumberParent` is returned, traversal of the Interrupt Service Tree continues with the parent member.

```
void InterruptEnabler   (InterruptSetMember  member,
                         void *              refCon);
```

The system does not use the passed values of `member` and `refCon`. These values are set and stored when the enabler is installed and passed when the enabler is flushed. Invoking `InterruptEnabler` will reenable the interrupt member's ability to propagate pending interrupts to the operating system.

```
InterruptSourceState InterruptDisabler
                        (InterruptSetMember  member,
                         void *              refCon);
```

The system does not use the passed values of `member` and `refCon`. These values are set and stored when the disabler is installed and passed when the disabler is flushed. Invoking `InterruptDisabler` will disable an interrupt member's ability to propagate pending interrupts to the operating system. On return, this routine returns the interrupt member's ability to propagate interrupts that were pending before this routine was invoked. A returned value of `SourceWasEnabled` means that the interrupt member's propagation state was enabled; a returned value of `SourceWasDisabled` means it was disabled.

Interrupt Set Creation and Options

The Macintosh system's 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 PCI bridges and multi-function cards.

The `CreateInterruptSet` function extends an IST.

```
OSStatus CreateInterruptSet(InterruptSetID         parentSet,
                            InterruptMemberNumber  parentMember,
                            InterruptMemberNumber  setSize,
                            InterruptSetID         setID,
                            InterruptSetOptions    options);

enum {
   kReturnToParentWhenComplete = 0x00000001,
   kParentDeMuxIsPolled        = 0x00000002
};
typedef OptionBits InterruptSetOptions;
```

`CreateInterruptSet` returns these error codes:

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

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 suceeded, and the variable pointed to by setID contains the member set ID of the new set's child members.

The options parameter operates this way:

- Option kReturnToParentWhenComplete sets a new default behavior for interrupt set completion. Any time a child in the set returns kIsrIsComplete (described in "Interrupt Return Values" on page 199), the dispatcher returns to the parent. A parent can thus get a second or third look at its children. If two children are simultaneously interrupting, the parent can recognize the second immediately.
- Option kParentDeMuxIsPolled tells the interrupt dispatcher to stop returning up the tree; that is, to short-circuit the search for a handler to take the interrupt. Its used in conjunction with the kIsrIsNotComplete return value.

GetInterruptSetOptions helps an expert determine how the interrupt dispatcher will handle an interrupt set, and ChangeInterruptSetOptions helps it change that behavior.

GetInterruptSetOptions

GetInterruptSetOptions returns the options for an interrupt set.

```
OSStatus GetInterruptSetOptions (InterruptSetID      set,
                                 InterruptSetOptions *options);
```

set	ID of the interrupt set
options	current options

DESCRIPTION

GetInterruptSetOptions returns in options the options for the interrupt set identified by set.

RESULT CODES

noErr	0	No error

ChangeInterruptSetOptions

ChangeInterruptSetOptions changes the options for an interrupt set.

```
OSStatus ChangeInterruptSetOptions (InterruptSetID      set,
                                    InterruptSetOptions *options);
```

`set`	ID of the interrupt set
`options`	new 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 PCI-to-PCI bridges 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.

RESULT CODES

`noErr`	0	No error

Control Routine Installation and Examination

The `InstallInterruptFunctions` function installs interrupt service routines in an interrupt member. It is described in "Interrupt Mechanisms" beginning on page 130.

The `GetInterruptFunctions` function fetches interrupt control routines installed in an interrupt member.

```
OSStatus
GetInterruptFunctions   (InterruptSetID         Set,
                         InterruptMemberNumber  Member,
                         void                   *refCon,
                         InterruptHandler       *Handler,
                         InterruptEnabler       *EnableFunc,
                         InterruptDisabler      *DisableFunc);
```

Pass the member set ID and the set member number in `Set` and `Member` to uniquely identify the interrupt member in the tree.

Upon successful completion of `GetInterruptFunctions`, the reference constant, the ISR, the IER, and the IDR are all returned to the caller.

`GetInterruptFunctions` returns these error codes:

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

Software Interrupts

The Driver Services Library provides several routines to create, run, and remove software interrupts. Some of these routines can be called only from certain execution levels, as described in "Device Driver Execution Contexts" beginning on page 174.

Note
Software interrupts cannot be used to allocate memory. ◆

```
TaskID CurrentTaskID (void)
```

`CurrentTaskID` returns the ID number of the currently running task. This routine can be called only from the noninterrupt execution level.

```
OSStatus CreateSoftwareInterrupt
      (SoftwareInterruptHandler  theHandler,
       TaskID                    theTask,
       void                      *theParameter,
       Boolean                   persistent,
       SoftwareInterruptID       *theSoftwareInterrupt)
```

`CreateSoftwareInterrupt` creates a software interrupt for a specified task. It can be called either from noninterrupt or secondary execution level.

`CreateSoftwareInterrupt` returns these error codes:

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

```
OSStatus SendSoftwareInterrupt
      (SoftwareInterruptID    theSoftwareInterrupt,
       void                   *theParameter)
```

`SendSoftwareInterrupt` runs a software interrupt task. It can be called from any execution level and acts as an asynchronous function.

`SendSoftwareInterrupt` returns these error codes:

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

Note
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. ◆

```
OSStatus DeleteSoftwareInterrupt
            (SoftwareInterruptID theSoftwareInterrupt)
```

`DeleteSoftwareInterrupt` removes a software interrupt. It can be called either from noninterrupt or secondary execution level.

`DeleteSoftwareInterrupt` returns these error codes:

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

Software interrupt handlers communicate by means of the Name Registry, described in Chapter 8. You can provide software interrupt communication in two ways:

- You can create a permanent software interrupt handler at noninterrupt level and store its `SoftwareInterruptID` value in the Registry. The main section of the driver can then retrieve the ID and run the handler, using `SendSoftwareInterrupt`. This technique does not queue interrupts, so the handler must be able to process multiple events.
- You can store the `taskID` value of the driver's control section in the Registry. The main driver section can then retrieve the value and use it to make temporary `CreateSoftwareInterrupt` and `SendSoftwareInterrupt` calls in pairs. This technique forces event queueing to the control section of the driver, so the handler processes one event per pair of calls. It allocates and frees system resources; therefore you must be prepared for error messages from `CreateSoftwareInterrupt` if system resources become exhausted.

Using these communication means, software interrupt services allow asynchronous operations between a controlling main driver section and a slave noninterrupt driver section.

Secondary Interrupt Handlers

Secondary interrupt handlers are the primary synchronization mechanism that a driver and its primary 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" beginning on page 174.

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 use 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.

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 following form:

```
typedef OSStatus (*SecondaryInterruptHandlerProc2) (void *1,
                                                    void *2);
```

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. Only one kind of secondary interrupt handler, that with two parameters, may be queued. You must specify the values of the two parameters at the time you queue the handler.

Secondary interrupts 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" beginning on page 174). You should make every attempt to limit the number of simultaneously queued secondary interrupt handlers because the memory resources available to them are limited.

Secondary interrupt handlers can be passed two parameters. Future versions of Mac OS may allow an exception handler to be associated with the interrupt handler; the `ExceptionHandler` parameter is currently ignored.

```
OSStatus QueueSecondaryInterruptHandler
               (SecondaryInterruptHandlerProc2  Handler,
                ExceptionHandler                ExceptionHandler,
                void                            *p1,
                void                            *p2);
```

`QueueSecondaryInterruptHandler` returns these error codes:

`noErr`	0	No error
`qErr`		

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. The secondary interrupt handler is invoked immediately in response to calls to this service; it is not queued.

```
OSStatus CallSecondaryInterruptHandler2
               (SecondaryInterruptHandlerProc2  Handler,
                ExceptionHandler                ExceptionHandler,
                void                            *p1,
                void                            *p2);
```

`CallSecondaryInterruptHandler2` returns these error codes:

`noErr`	0	No error
	–1	Call failed

Code Example

The code sample in Listing 9-1 shows typical interrupt registration during driver initialization.

Listing 9-1 Interrupt code sample

```
#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;

InterruptMemberNumber
myISRHandler(  InterruptSetMember   member,
               void *               refCon )
   {

   // Do what ever is required to service your hardware here.

   return kIsrIsComplete;
   }

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`, the units required by the timing services. You may use conversion routines to convert from `Nanoseconds` or `Durations` into the `AbsoluteTime` system units. This conversion can introduce errors, but errors are typically limited to one unit of the time base.

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 the following service:

```
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

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, are returned. If you multiply an `AbsoluteTime` value by `theAbsoluteTimeToNanosecondNumerator`

and divide the result by the value of `theAbsoluteTimeToNanosecondDenominator`, the result is nanoseconds.

The `minAbsoluteTimeDelta` is the minimum number of `AbsoluteTime` units that can change at any given time. For example, if the Power Macintosh hardware changes the decrementer in quantities of 128, then the `minAbsoluteTimeDelta` returned by `TimeBaseInfo` would be 128.

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 Ma 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 `Durations`.

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 Macintosh System 7 Time Manager—positive values are in units of milliseconds, negative values are in units of microseconds. Therefore the value 1500 is 1,500 milliseconds or 1.5 seconds while the value –8000 is 8,000 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 nor 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 `duration` are reserved and have special meaning. The value 0 specifies no duration. The value 0x7FFFFFFF, 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" on page 208, 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.

```
AbsoluteTime UpTime (void);
```

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 LaterAbsoluteTime,
          AbsoluteTime EarlierAbsoluteTime);

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 LaterAbsoluteTime,
          AbsoluteTime EarlierAbsoluteTime);

Duration AbsoluteDeltaToDuration
         (AbsoluteTime LaterAbsoluteTime,
          AbsoluteTime EarlierAbsoluteTime);
```

Note
If you subtract an `EarlierAbsoluteTime` value from a `LaterAbsoluteTime` value, the result is 0, not a negative number. ◆

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. You can set an interrupt timer from any driver execution context.

```
typedef KernelID TimerID;
```

```
OSStatus SetInterruptTimer
         (AbsoluteTime                *expirationTime,
          SecondaryInterruptHandler   Handler,
          void *                      p1,
          TimerID *                   Timer);
```

The parameter `expirationTime` is the absolute time at which the timer will expire.

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`.

```
OSStatus DelayFor (Duration expirationTime);
```

Parameter `expirationTime` is the amount of time to delay, expressed as a positive number in milliseconds or as a negative number in microseconds. `DelayFor` is not available at the hardware interrupt level.

`DelayFor` returns these error codes:

`noErr`	0	No error
	–1	Call failed

```
OSStatus DelayForHardware (AbsoluteTime absoluteTime);
```

Parameter `absoluteTime` is the amount of time to delay in processor-dependent units. You can call `NanosecondsToAbsolute` to obtain timing for the current PowerPC processor. `DelayFor`Hardware may be called at the hardware interrupt level.

`DelayForHardware` returns these error codes:

`noErr`	0	No error
	–1	Call failed

Interrupt timers consume memory resources from the time they are invoked until the time they expire or are cancelled. They do this regardless of the execution context (see “Device Driver Execution Contexts” beginning on page 174). You should make every attempt to limit the number of interrupt timers because the memory resources available to them are limited.

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.

```
OSStatus CancelTimer (TimerID Timer, AbsoluteTime *TimeRemaining);
```

`CancelTimer` cancels a previously created timer. It returns in `TimeRemaining` the amount of time that was left in the timer when it was cancelled.

An error is returned if the timer has either already expired or been canceled.

With the current version of 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 cancelled before the other secondary handler runs.

Atomic Memory Operations

The Driver Services Library provides the following 32-, 16-, and 8-bit atomic memory operations for use by device drivers.

```
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 );

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 );

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 );
```

The atomic routines perform various operations on the memory address specified by `value`:

- `IncrementAtomic` increments the value by one and `DecrementAtomic` decrements it by one. 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.
- 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.

These routines take logical address pointers and ensure that the operations are atomic with respect to all devices (for example, other processors, DMA engines) that participate in the coherency architecture of the Macintosh system.

```
TestAndSet     (UInt32  theBit
                UInt8   *theAddress);

TestAndClear   (UInt32  theBit
                UInt8   *theAddress);
```

`theBit` The bit number in the range 0 through 7.

`theAddress` The address of the byte in which the bit is located.

`TestAndSet` and `TestAndClear` set and clear a single bit in a byte at a specified address.

Queue Operations

The Driver Services Library provides the following I/O Parameter Block queue manipulation functions:

```
OSErr PBQueueInit (QHdrPtr qHeader);
OSErr PBQueueCreate (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);
```

The first three functions listed above cannot be called from interrupt handlers. For a list of functions that can be called from various interrupt levels, see "Service Limitations" beginning on page 216.

String Operations

The following C and Pascal string manipulation functions are available to drivers:

```
StringPtr   PStrCopy (StringPtr dst, const StringPtr src);
char        *CStrCopy (char *dst, const char *src);
```

`PStrCopy` 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.

```
StringPtr PStrNCopy
   (StringPtr dst, const StringPtr src, uint32 max);
char *CStrNCopy (char *dst, const char *src, uint32 max);
```

`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`, the `dst` string will be padded with null characters. If `src` string is longer than `max` the `dst` string will not be null-terminated.

```
StringPtr   PStrCat (StringPtr dst, const StringPtr src);
char        *CStrCat (char *dst, const char *src);
```

`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.

```
StringPtr PStrNCat
   (StringPtr dst, const StringPtr src,uint32 max);
char *CStrNCat (char *dst, const char *src, uint32 max);
```

`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`.

```
short PStrCmp (const StringPtr s1, const StringPtr s2);
short CStrCmp (const char *s1, const char *s2);
```

`PStrCmp` and `CStrCmp` compare the Pascal and C strings `s1` and `s2` 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.

```
short PStrNCmp
   (const StringPtr s1, const StringPtr s2, uint32 max);
short CStrNCmp (const char *s1, const char *s2, uint32 max);
```

`PStrNCmp` and `CStrNCmp` compare the first `max` C and Pascal strings `s1` and `s2` 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.

```
uint32 PStrLen (const StringPtr src);
uint32 CStrLen (const char *src);
```

`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.

```
void PStrToCStr (char *dst, const Str255 src);
void CStrToPStr (Str255 dst, const char *src);
```

`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);`	Enters the system debugger.
`void SysDebugStr (StringPtr string);`	Enters the system debugger and displays the Pascal string.

Service Limitations

This section lists the driver services available for Primary and Secondary Interrupt levels, as described in "Device Driver Execution Contexts" beginning on page 174. 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.

Services That Can Be Called From the Primary Interrupt Level

`QueueSecondaryInterruptHandler`
`UpTime`
`SetInterruptTimer`
`PBEnqueue`
`RegistryPropertyGetSize`
`RegistryPropertyGet`
`RegistryPropertySet`
`SendSoftwareInterrupt`
String functions
Debugger routines
`BlockCopy`
Atomic operations

Services That Can Be Called From Secondary Interrupt Handlers

`CallSecondaryInterruptHandler2`
`QueueSecondaryInterruptHandler`
`UpTime`
`SetInterruptTimer`
`CancelTimer`
`PoolDeallocate`
`CheckpointIO`
`RegistryPropertyGetSize`
`RegistryPropertyGet`
`RegistryPropertySet`
`PBEnqueue`
`PBEnqueueLast`
`PBDequeue`
`PBDequeueFirst`
`PBDequeueLast`
`CreateSoftwareInterrupt`
`SendSoftwareInterrupt`
String functions
Debugger routines
`BlockCopy`
Atomic operations

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" briefly summarizes the conformance of expansion ROMs on Macintosh-compatible PCI cards with the PCI specification.
- "The Device Tree" describes the configuration data structure that the Macintosh firmware creates during the Open Firmware startup process.
- "Typical NVRAM Structure," beginning on page 221, illustrates how nonvolatile RAM is allocated in a typical Power Macintosh computer.
- "PCI Non-Memory Space Cycle Generation," beginning on page 224, lists routines that you can use to access memory in the PCI I/O address space.
- "Card Power Controls," beginning on page 235, describes calls that Mac OS uses to control PCI card power levels.
- "Macintosh System Gestalt" beginning on page 236, lists the gestalt calls that drivers and card firmware can make to determine the hardware features and system software present in the user's computer.

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.

The Device Tree

The device tree is a data structure that the Macintosh startup firmware creates in system RAM to provide information about configured PCI devices to other software, including firmware on PCI cards. Attached to it are the drivers and support software that PCI 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.

The device tree is the structure from which Mac OS extracts the original information to create the device portion of the NameRegistry, described in Chapter 8, "Macintosh Name Registry." A device tree entry may be a device entry (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 device entry called the *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.

The Expansion Bus Manager includes the following call to let drivers and card firmware convert physical device addresses to the logical addresses required by most of the Macintosh support software for PCI Cards:

```
OSStatus DeviceAddressPhysicalToLogical
      (RegPropertyValue       devicePhysicalAddress,
       RegPropertyValueSize   devicePhysicalAddressSize,
       LogicalAddress         *deviceLogicalAddress);
```

Nonvolatile RAM

Power Macintosh computers that support the PCI bus contain at least 4 KB of ***nonvolatile RAM (NVRAM).*** The NVRAM chips are flash ROM, or RAM powered by the computer's local battery, so that they retain 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 is shown in Table 10-1.

Table 10-1 Typical NVRAM space allocations

Length, bytes	Description
4096	Operating system partition
768	Reserved by Apple
256	Reserved by Apple (Macintosh PRAM)
3072	Open Firmware partition

The allocations shown in Table 10-1 provide permanent configuration data storage, both for the Macintosh system 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 1024 bytes of NVRAM space for use by Macintosh firmware and system software. 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 3072 bytes of NVRAM space might typically be used by the Open Firmware startup process to support PCI expansion cards.

If the `kRegPropertyValueIsSavedToNVRAM` modifier of a property entry is set, the contents of that property entry will be preserved in NVRAM. During 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" on page 159.

When a PCI card is first detected during the Open Firmware startup process, the Macintosh firmware stores its vendor and device IDs in NVRAM. During subsequent startups, if the actual card in a slot does not correspond to the stored data the Macintosh firmware updates the stored data and deletes the configuration data for that slot.

The `little-endian?` variable, discussed in "Addressing Mode Determination" on page 20, is stored in the Open Firmware NVRAM space.

Using NVRAM to Store Device Properties

NVRAM can be used to store device properties in a permanent way. However, such storage is necessary only for devices used during the Open Firmware startup process,

because other devices can store an unlimited amount of permanent information in the system preferences file.

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. The Name Registry uses the following format to store the property:

- device location (4 bytes), an absolute location within the PCI system hardware universe. It corresponds to the slot ID in NuBus systems. The format of this value is not public and its value is not visible to the driver function.
- 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 use the C/F Registration Requests Hypercard stack to register a creator ID. The stack sends an AppleLink message to Apple Developer Technical Support, which registers your request and replies with a confirmation message. You do not need to be an an Apple Partner or Associate to make use of this service.
- property value (8 bytes max), a value is stored by `RegistryPropertySet` or `RegistryPropertyCreate` and retrieved by `RegistryPropertyGet`.

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 8 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 eight bytes in an NVRAM property.
- Specifies a property name that longer than 4 bytes.

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.

PCI Non-Memory Space Cycle Generation

"PCI Host Bridge Operation," beginning on page 8, 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 non-memory access routines use the type `RegEntryIDPtr` to identify device nodes in the device tree, as described in Chapter 8, "Macintosh 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.

I/O Space Cycle Generation

The Expansion Bus Manager includes six routines that let you read and write data to specific I/O addresses. They are described in this section.

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 *value);
```

`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 address to be accessed. Its value is the sum of the base address of the device plus the offset to the desired I/O address.

`value` 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 value in `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read byte
`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 *value);
```

`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 address to be accessed. Its value is the sum of the base address of the device plus the offset to the desired I/O address.
`value`	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 `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read word
`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 *value);
```

`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 address to be accessed. Its value is the sum of the base address of the device plus the offset to the desired I/O address.

`value` 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 `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read long word
`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 address to be accessed. Its value is the sum of the 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
`Err`	–1	Unable to write byte
`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 address to be accessed. Its value is the sum of the 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
`Err`	–1	Unable to write word
`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 address to be accessed. Its value is the sum of the 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
`Err`	–1	Unable to write long word
`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 8, "Macintosh 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 *value);
```

`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).
`value`	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 `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read byte
`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 *value);
```

`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).
`value`	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 `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read word
`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 *value);
```

`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).

`value` 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 `value`.

RESULT CODES

`noErr`	0	No error
`Err`	–1	Unable to read long word
`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
`Err`	–1	Unable to write byte
`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
`Err`	–1	Unable to write word
`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
`Err`	–1	Unable to write long word
`deviceTreeInvalidNodeErr`	–2538	Device node not in the device tree

Interrupt Acknowledge Cycle Generation

The routines described in this section generate the interrupt acknowledge cycles on the PCI bus required to access PCI interrupt acknowledge space.

ExpMgrInterruptAcknowledgeReadByte

You can use the `ExpMgrInterruptAcknowledgeReadByte` function to read the byte value at a specific address in PCI interrupt acknowledge space.

```
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 at a specific address in PCI interrupt acknowledge space.

```
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 at a specific address in PCI interrupt acknowledge space.

```
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.

ExpMgrSpecialCycleBroadcastLong

You can use the `ExpMgrSpecialCycleBroadcastLong` function to broadcast the long word value in `value` on the PCI bus.

```
OSErr ExpMgrSpecialCycleBroadcastLong(RegEntryIDPtrentry,
                                      UInt32        value);
```

`entry` Pointer to a Name Registry entry ID

`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.

```
OSErr ExpMgrSpecialCycleWriteLong   (RegEntryIDPtr entry,
                                     UInt32        value);
```

`entry` Pointer to a Name Registry entry ID

`value` The value to be written

BlockMove Routines

The system software for PCI-based Power Macintosh computers includes new extensions to the `BlockMove` routine that is described in *Inside Macintosh: Memory.* The extensions provide improved performance for software running in native mode.

The new `BlockMove` extensions provide several benefits for developers.

- They're optimized for the PowerPC 603 and PowerPC 604 processors, rather than the PowerPC 601.
- They're compatible with the new dynamic recompilation emulator.
- They provide a way to handle cache-inhibited address spaces.
- They include new high-speed routines for setting memory to zero.

The new `BlockMove` extensions do not use the string instructions, which are fast on the PowerPC 601 but slow on other PowerPC implementations.

Table 10-2 lists the different versions of the `BlockMove` function and indicates for each one whether it can be used with buffers or other uncacheable destination locations.

Table 10-2 BlockMove versions

BlockMove version	Can be used with buffers
`BlockMove`	No
`BlockMoveData`	No
`BlockMoveDataUncached`	Yes
`BlockMoveUncached`	Yes
`BlockZero`	No
`BlockZeroUncached`	Yes

Except for `BlockZero` and `BlockZeroUncached`, the new `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 `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. ▲

An easy way to use the most appropriate `BlockMove` version is to call the `BlockCopy` routine described on page 190. `BlockCopy` calls the `BlockMove` version that works most efficiently for the current task and hardware configuration.

Accessing Device Registers

The PCI property `"assigned-addresses"` provides vector entries into devices on expansion cards. Apple has added another property, `"AAPL,address"`, that provides a vector of 32-bit values, where the nth value corresponds to the nth assigned-addresses vector entry. You can use the `"AAPL,address"` property as a pointer to a device's registers. If the `"AAPL,address"` entry corresponds to an address in PCI I/O space, the system hardware will execute I/O cycles on the PCI bus. The `"assigned-addresses"` and `"AAPL,address"` properties are listed in Table 8-1 on page 156.

To access a register on a device using these properties, 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 variabler 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.

Card Power Controls

It is desirable for every PCI card to be able to operate in a low-power mode, so it will conform to energy-saving system standards. If a PCI expansion card normally consumes more than 3 A at 5 V or 2 A at 3.3 V, the low-power capability is particularly important. The Macintosh firmware and system software control the card's power level by sending a control call with `csCode` = $47 (`driverPowerLow`) to set the power level low, or

`csCode` = $48 (`driverPowerHigh`) to set the power level high. These commands may be sent at any time.

A card's driver may elect to ignore power switching commands, returning an error value of `driverPowerError`. If a switch from high-power to low-power might interrupt a current operation, the card's driver may return a `driverPowerSwitchAbort` error. This error return also asks the Macintosh system not to reduce power to other services.

Macintosh System Gestalt

When it builds the device tree, the Open Firmware code in 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" beginning on page xxiii. The available properties are shown in Table 10-3.

Table 10-3 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
`"pgsz"`	Logical page size
`"proc"`	Microprocessor type
`"prty"`	Parity attributes
`"ram "`	Physical RAM size
`"rom "`	System ROM size
`"romv"`	System ROM version
`"ser "`	Serial hardware attributes
`"snd "`	Sound attributes
`"tv  "`	TV attributes
`"vers"`	Gestalt version
`"vm  "`	Virtual memory attributes

Note
Specific Macintosh computer models may lack some of the gestalt values listed in Table 10-3, so the corresponding properties will not appear in the gestalt node. ◆

PCI expansion card firmware can explore the gestalt node to determine the hardware and firmware environment available to it. For example, Listing 10-1 shows typical code to extract the 32-bit value of the Macintosh virtual memory attributes from the `"vm  "` property of the gestalt node.

Listing 10-1 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);
```

Graphics Drivers

This chapter discusses the control and status calls required to implement a PCI graphics or video display driver for Mac OS on the second generation of Power Macintosh computers.

Display drivers are currently generic (category `'ndrv'`) and have a service type of video (type `'vide'`). They export a Driver Description structure and use the `DoDriverIO` entry point. For specific information about generic native drivers, see Chapter 7, "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" beginning on page xxiii.

Apple has revised the way that Macintosh computers automatically sense monitor characteristics. For more information see "Display Timing Modes," beginning on page 259, and *Macintosh New Technical Notes HW-30,* available from Apple Developer Support.

Typical Driver Description

A typical driver description structure for a PCI graphics card driver looks like the following:

```
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 Runtime Requirements
   {
      kdriverIsUnderExpertControl   // Runtime options
      + kdriverIsOpenedUponLoad,
      "\p.Display_Video_Apple_Viper",
   },

   // OS Runtime Info
   {
      1,                         // Number of service categories
```

```
      {
         kServiceCategoryndrvdriver,// We support 'ndrv' category
         kNdrvTypeIsVideo,          // Video type

   // Version of service
         1, 0, 0, 0                 // Major, minor, stage, rev
      }
   }
};
```

Note
For the Display Manager to load and install a driver, the runtime 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 runtime service category and `kNdrvTypeIsVideo` as the type within the category. ◆

Graphics Driver Routines

In the past, graphics drivers and the MacOS relied on a card's NuBus declaration ROM to get information on the card's capabilities. In the second generation of Power Macintosh computers, the programming interface for graphics drivers has been revised to let drivers provide the same information. MacOS has also been revised to fetch this information from drivers instead of from a card's ROM.

This section details the control and status calls that a graphics driver must respond to.

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 = 0x0)

The `cscReset` call resets the video card to its startup state, which should include its default pixel depth and default colors.

```
err = Control(theDeviceRefNum, cscReset, &theVDPageInfo );
<--      csMode         Relative bit depth after reset
 --      csData         Unused
<--      csPage         Page after reset
<--      csBaseAddress  Base address of video RAM
```

If the card supports multiple video pages in the default mode, page 0 should be switched in. The driver should also reinitialize its private storage, including areas for returned parameters.

KillIO (csCode = 0x1)

The `cscKillIO` call stops any I/O requests currently being processed and removes any pending I/O requests. If the card does not support asynchronous calls, this routine may return a `NoErr` code.

SetMode (csCode = 0x2)

The `cscSetMode` call sets the pixel depth of the screen.

```
err = Control(theDeviceRefNum, cscSetMode, &theVDPageInfo );
-->      csMode         Desired relative bit depth
--       csData         Unused
-->      csPage         Desired display page
<--      csBaseAddress  Base address of video RAM for this csMode
```

Too improve the screen appearance during mode changes, devices with settable color tables should set all entries of the color lookup table (CLUT) to 50% gray. If the video card supports 16-bit or 32-bit pixel depths, this routine should set an internal flag to indicate direct mode operations.

SetEntries (csCode = 0x3)

If the video card is an indexed device, the optional `cscSetEntries` control routine should change the contents of the card's CLUT.

```
err =
Control(theDeviceRefNum, cscSetEntries, &theVDSetEntryRecord );
-->      csTable  Pointer to ColorSpec array
-->      csStart  firstEntry in Table
-->      csCount  number of entries to set
```

If the card does not have a CLUT, it will never receive a `cscSetEntries` call. 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 zero-based; their values are 1 less than the desired amount. For a description of a CLUT,

refer to the information on Color QuickDraw in *Inside Macintosh: Imaging With QuickDraw.* The `cscSetEntries` control call is also described in the Color Manager chapter of the same book.

If the video card is a direct device, the system should never issue this call, but if it does, `cscSetEntries` should return an error indication. If a direct device contains CLUT hardware, the `cscGrayPage` and `cscSetGamma` are responsible for initializing the hardware properly.

In the 16-bit and 32-bit video modes associated with direct devices, the display color is implied directly by the pixel value. Logically, the three digital-to-analog converter (DAC) channels in the hardware are completely independent and are assumed to be ascending linear ramps in all channels. Since the effect of the `cscSetEntries` routine (in the Color Manager) is to modify the QuickDraw drawing environment, the `cscSetEntries` call has no meaning for a direct device.

Note
The `cscSetEntries` control routine is a low-level function and should only be used in special cases. You may find it easier to implement an equivalent higher-level call. Refer to Device Manager and Color Manager information in *Inside Macintosh: Imaging With QuickDraw* for details of alternate function calls. ◆

SetGamma (csCode = 0x4)

The optional `cscSetGamma` control routines sets a gamma table in the driver that corrects RGB color values.

```
err = Control(theDeviceRefNum, cscSetGamma, &theVDGammaRecord );
-->      csGTable    Pointer to gamma table
```

The gamma table compensates for nonlinearities in a displays's color response by providing either a function or a lookup value that associates each displayed color with an absolute RGB value. The gamma table is described with the graphics devices information in *Inside Macintosh: Imaging With QuickDraw.*

To reduce visible flashes resulting from color table changes, the `cscSetGamma` call works in conjunction with a `cscSetEntries` call on indexed devices. The `cscSetEntries` call first loads new gamma correction data into the driver's private storage, and then the next `cscSetEntries` call applies the gamma correction as it changes the CLUT. If the hardware performs gamma correction externally to the CLUT hardware, then the `cscSetGamma` call should take effect immediately. The `cscSetGamma` calls are always followed by `cscSetEntries` calls.

For direct devices, the `cscSetGamma` routine first sets up the gamma correction data table. Next, it synthesizes a black-to-white linear ramp in RGB. Finally, it applies the new gamma correction to the ramp and sets the data directly in the hardware. Proper gamma

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.

GrayPage (csCode = 0x5)

The optional `cscGrayPage` control routine should fill the specified video page with a dithered gray pattern in the current video mode. The page number is zero-based.

```
err = Control(theDeviceRefNum, cscGrayPage, &theVDPageInfo );
--          csMode         Unused
--          csData         Unused
-->         csPage         Desired display page to gray
--          csBaseAddress  Unused
```

The purpose of this routine is to eliminate visual artifacts on the screen during mode changes. When an application changes the screen depth, the contents of the frame buffer immediately acquire a new color meaning. To avoid annoying color flashes, the `cscSetMode` control call (first in the depth change sequence) sets the entire contents of the CLUT to 50% gray, so that all possible indexes in either the old or new depth appear the same. This routine is called to fill the frame buffer with the new 50% dither pattern. In the last step of the mode change sequence the color table is filled, making the 50% dither pattern visible.

For direct video devices, `cscGrayPage` also builds a three-channel linear gray color table; after the table has been gamma corrected, it loads it into the color table hardware. Since 32-bit QuickDraw is the only QuickDraw present, the base address will be a 32-bit `address#` value. To simplify the code, you should always write `cscGrayPage` to the screen in 32-bit addressing mode.

SetGray (csCode = 0x6)

The optional `cscSetGray` control routine is used with indexed devices to determine whether the control routine with `cscSetEntries` fills a card's CLUT with actual colors or with the luminance-equivalent gray tones.

```
err = Control(theDeviceRefNum, cscSetGray, &theVDGrayRecord );
-->      csMode      Colors or grays
```

For actual colors, the control routine is passed a `csMode` value of 0; for gray tones 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 the `cscSetGray` call is issued to a direct device, the device sets the internal mapping state flag and returns a `CtlGood` result but does not luminance-map the color table. Short of using the control routine `cscDirectSetEntries`, there is no way to preview luminance-mapped color images on the color display of a direct device.

SetInterrupt (csCode = 0x7)

The optional `cscSetInterrupt` routine controls the generation of vertical blanking (VBL) interrupts.

```
err = Control(theDeviceRefNum, cscSetInterrupt, &theVDPageInfo );
-->         csMode         Enable or Disable Interrupts
--          csData         Unused
--          csPage         Unused
--          csBaseAddress  Unused
```

To enable interrupts, pass a `csMode` value of 0; to disable interrupts, pass a `csMode` value of 1.

DirectSetEntries (csCode = 0x8)

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 `cscDirectSetEntries` routine provides the direct device with indexed mode functionality identical to the regular `cscSetEntries` call.

```
err = Control(theDeviceRefNum, cscDirectSetEntries,
      &theVDSetEntryRecord);
-->      csTable  Pointer to ColorSpec array
-->      csStart  firstEntry in Table
-->      csCount  number of entries to set
```

The `cscDirectSetEntries` routine has exactly the same functions and parameters as the regular `cscSetEntries` routine, but it works only on a direct device. If this call is issued to an indexed device, it should return a `CtlBad` error indication.

Note
The application that calls the `cscDirectSetEntries` routine is responsible for restoring the triple-linear-ramp direct-color environment when it completes the color table animation. ◆

The `directSetEntries` routine is implemented separately from the `SetEntries` routine to prevent applications that get direct access to the driver from indiscriminately changing the hardware and rendering the system unusable.

SetDefaultMode (csCode = 0x9)

The `cscSetDefaultMode` control call is obsolete for graphics drivers in the second generation of Power Macintosh computers. A driver can respond with the `cscSavePreferredConfiguration` control routine described in "SavePreferredConfiguration (csCode = 0x10)" on page 248.

SwitchMode (csCode = 0xA)

The `cscSwitchMode` routine is required in drivers that support the Display Manager.

```
err = Control(theDeviceRefNum, cscSwitchMode,
       &theVDSwitchInfoRecord );
-->      csMode         Relative Bit Depth to switch to
-->      csData         DisplayModeID to switch into
-->      csPage         Video Page number to switch into
<--      csBaseAddress  Base address of the new display mode
```

The `VDSwitchInfoRec` structure, described on page 266, indicates what depth mode to switch to, the `DisplayModeID` 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`.

After switching modes, the driver should store that mode in parameter RAM and use that mode as the startup mode.

SetSync (csCode = 0xB)

`SetSync` complements `GetSync`, described on page 252. 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`:

```
err = Control(theDeviceRefNum, cscSetSync, &theVDSyncInfoRec);
```

Following is the information that the status routine must return in the fields of the `VDSyncInfoRec` record (defined on page 252) passed by `SetSync`:

→ `csMode` Bit map of the sync bits that need to be disabled or enabled.

↔ `csFlag` As an input, contains 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.

As an output, the driver should clear the `csFlag` field and set any bits according to the following bit definition:

`kVRAMNotAccessible = 7:` Set if VRAM cannot be accessed while the syncs are off.
`kVRAMNeedsUpdate = 6:` Set if VRAM must be updated after syncs are turned on.

To preserve some 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 11-1 illustrates the relationship between modes and signals.

Table 11-1 Implementing VESA DPMS modes with SetSync

Mode	Horizontal sync	Vertical sync	Video	Powqer 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 the only the composite sync line, only the On and Off power saving modes are possible.

SavePreferredConfiguration (csCode = 0x10)

`SavePreferredConfiguration` complements the `GetPreferredConfiguration` control routine described on page 255. The two resemble `SetDefaultMode` and `GetDefaultMode`, with the exception that `SavePreferredConfiguration` takes a `VDSwitchInfoRec structure` (described on page 266) instead of an unsigned `char`. 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 the second generation of Power Macintosh computers, see "Typical NVRAM Structure" beginning on page 221.

```
err = Control
   (theDeviceRefNum, cscGetPreferredConfig, &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
`--`	`csBaseAddress`	Unused

The `csBaseAddress` parameter is 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. ◆

DriverPowerLow (csCode = 0x2F)

The `cscDrivePowerLow` call switches the display device to low power mode. It is a generic call that is added to the `SetSync` call described on page 259. The driver should return `noErr`.

DriverPowerHigh (csCode = 0x30)

The `cscDrivePowerHigh` call switches the display device to high power mode. It is a generic call that is added to the `SetSync` call described on page 259. The driver should return `noErr`.

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 = 0x2)

The `cscGetMode` routine is required. It must return the current video mode, page, and base address.

```
err = Status(theDeviceRefNum, cscGetMode, &theVDPageInfo );
<--      csMode         Current relative bit depth
 --      csData         Unused
<--      csPage         Current display page
<--      csBaseAddress  Base address of video RAM for this csMode
```

GetEntries (csCode = 0x3)

The `cscGetEntries` routine is required. It must return the specified number of consecutive CLUT entries, starting with the specified first entry.

```
err =
Status(theDeviceRefNum, cscGetEntries, &theVDSetEntryRecord );
<->      csTable  pointer to ColorSpec array
-->      csStart  firstEntry in Table
-->      csCount  number of entries to set
```

If gamma table correction is used, the values returned may not be the same as the values orignally passed by `cscSetEntries`. 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 zero-based; their values are one less than the desired amount.

Although direct video modes do not have logical color tables, the `cscGetEntries` status routine should continue to return the current contents of the CLUT, just as it would in an indexed mode.

The `cscGetEntries` control routine is a low-level function and should only be used in special cases. You may find it easier to implement an equivalent higher-level call. Refer to Device Manager and Color Manager information in *Inside Macintosh: Imaging with Quickdraw* for information about alternate function calls.

GetPages (csCode = 0x4)

The `cscGetPages` routine is required. It must return 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 zero-based.

```
err = Status(theDeviceRefNum, cscGetPages, &theVDPageInfoRecord );
 --      csMode         Unused
 --      csData         Unused
<--      csPage         Number of Display Pages available
 --      csBaseAddress  Unused
```

GetBaseAddress (csCode = 0x5)

The `cscGetBaseAddress` routine is required. It must return the base address of a specified page in the current mode.

```
err = Status(theDeviceRefNum, cscGetBaseAddr, &theVDPageInfo );
 --      csMode         Unused
 --      csData         Unused
 -->     csPage         Desired Page
 <--     csBaseAddress  Base Address of VRAM for the desired page
```

The `cscGetBaseAddress` routine allows video pages to be written to even when not displayed.

GetGray (csCode = 0x6)

The `cscGetGray` routine is required. It must return a value indicating whether the `cscSetEntries` routine has been conditioned to fill a card's CLUT with actual color or with the luminance-equivalent gray tones.

```
err = Status(theDeviceRefNum, cscGetGray, &theVDGrayRecord );
 <--      csMode      Colors or grays
```

For actual colors (the default case), the value return `csMode` is 0; for gray tones it is 1. The value returned can be set by a control call with `csCode` = 0x6.

GetInterrupt (csCode = 0x7)

The optional `cscGetInterrupt` status routine returns a value of 0 if VBL interrupts are enabled and a value of 1 if VBL interrupts are disabled.

```
err = Status(theDeviceRefNum, cscGetInterrupt, &theVDPageInfo );
<--       csMode          Interrupts enabled or disabled
--        csData          Unused
--        csPage          Unused
--        csBaseAddress   Unused
```

GetGamma (csCode = 0x8)

The `cscGetGamma` routine returns a pointer to the current gamma table.

```
err = 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 = 0x9)

The `cscGetDefaultMode` control call is obsolete for PCI graphics drivers. You can respond with the `cscGetPreferredConfiguration` control routine described on "GetPreferredConfiguration (csCode = 0x10)" on page 255.

GetCurrentMode (csCode = 0xA)

The `GetCurrentMode` call uses a `VDSwitchInfoRec` structure. Instead of returning a `slotID` (or `spID`) in the `csData` field, PCI graphics drivers return a `DisplayModeID`.

```
err = Status (theRefNum, cscGetCurMode, &theVDSwitchInfoRec );
```

The Display Manager can use this routine to get information about the current mode for the display card. Following is the information that the status routine must return in the fields of the `VDSwitchInfoRec` record passed by `GetCurrentMode`:

<--	`csMode`	current relative bit depth
<--	`csData`	`DisplayModeID` of current resolution
<--	`csPage`	current page
<--	`csBaseAddr`	base address of current page
<--	`csReserved`	reserved or set to 0

GetSync (csCode = 0xB)

The use of the `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 Device Power Management Standard (DPMS) as well as enable a sync-on-green mode for the frame buffer.

A new `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
};
```

The `csFlags` parameter is used by drivers to convey information about hardware requirements that are needed when implementing `SetSync`. Two bits are defined:

```
enum {
   kVRAMNotAccessible = 7,//  Set if VRAM cannot be accessed while the
                          //  horizontal/vertical/composite syncs are disabled.
   kVRAMNeedsUpdate = 6   //  Set if VRAM needs to be updated after the
                          //  horizontal/vertical/composite syncs are enabled.
};
```

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 listed on page 252.

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 = 0xC)

The `cscGetConnection` routine is used by the Display Manager to gather information about the attached display.

```
err = 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
<--      csConnectReserved    reserved
```

See “Responding to cscGetConnectionInfo” beginning on page 260 for more information on how to implement the `cscGetConnection` routine.

GetModeTiming (csCode = 0xD)

The `cscGetModeTiming` routine is required to report timing information for the desired `displayModeID`.

```
err =
Status(yourDeviceRefNum, cscGetModeTiming, &theVDTimingInfoRec);

-->     csTimingMode      Desired DisplayModeID
--      csTimingReserved  Unused
<--     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" beginning on page 259 for more details on the `cscGetModeTiming` routine.

GetModeBaseAddress

The `cscGetModeBaseAddress` call is obsolete in the second generation of Power Macintosh computers.

GetPreferredConfiguration (csCode = 0x10)

`GetPreferredConfiguration` complements `SavePreferredConfiguration`, described on page 248. These two calls are similar to the previous `SetDefaultMode` and `GetDefaultMode` calls, except that `GetPreferredConfiguration` passes a `VDSwitchInfoRec` structure (described on page 266) instead of an unsigned `char`. `GetPreferredConfiguration` returns the data that was set using `SavePreferredConfiguration`.

```
err = Status
   (theDeviceRefNum, cscGetPreferredConfig, &theVDSwitchInfo);
```

The Monitors control panel can use this call to get the graphics driver's preferred configuration and update the resolution list displayed to the user. Following is the information that the driver's status routine must return in the fields of the `VDResolutionInfoRec` record passed by `GetPreferredConfiguration`:

<--	`csMode`	Relative bit depth of preferred resolution
<--	`csData`	`DisplayModeID` of preferred resolution
--	`csBaseAddress`	Unused

GetNextResolution (csCode = 0x11)

The previous NuBus-based Display Manager and Monitors control panel used enabled and disabled functional sResources to build a list of available resolutions for the current display. For example, a multiple-scan display may have available resolutions of 640 by 480 pixels at 67 Hz and 832 by 624 pixels at 75 Hz. PCI-based graphics cards cannot use the previous mechanism because they don't have NuBus declaration ROMs. The `GetNextResolution` call provides the same information from PCI cards.

```
err = Status
(theDeviceRefNum, cscGetNextResolution, &theVDResolutionInfoRec);
```

The Monitors control panel can use `cscGetNextResolution` to get the resolutions supported by the driver. Following is the information that the status routine must return in the fields of the `VDResolutionInfoRec` record passed by `GetNextResolution`:

`-->`	`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" beginning on page 265.

GetVideoParameters (csCode = 0x12)

PCI graphics drivers must return video parameter information to their callers by responding to `GetVideoParameters`.

```
err = Status (theDeviceRefNum, cscGetVideoParameters,
             &theVDVideoParametersRec);
```

The Monitors control panel can use this routine to get the pointer to the Video Parameter Block for a particular bit depth. Following is the information that the status routine must return in the fields of the `VDResolutionInfoRec` record (defined on page 267) passed by `GetVideoParameters`:

`-->`	`csDisplayModeID`	ID of the desired Mode
`-->`	`csDepthMode`	Relative bit depth for the Display Mode
`<->`	`*csVPBlockPtr`	Pointer to a `VPBlock`
`<--`	`csPageCount`	Vertical Refresh Rate of the Screen
`<--`	`csDeviceType`	Direct, fixed, or CLUT

The `cscGetVideoParameters` routine accepts a `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`.

GetGammaInfoList (csCode = 0x14)

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.

```
err = Status
   (theDeviceRefNum, cscGetGammaInfoList, &theVDGammaListRec);
```

Following is the information that the status routine must return in the fields of the `VDGetGammaListRec` record passed by `cscGetGammaInfoList`:

`-->`	`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 215.

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 = 0x15)

The `RetrieveGammaTable` routine copies the designated gamma table into the designated location.

```
struct VDRetrieveGammaRec {
   GammaTableID csGammaTableID;  // ID of gamma table to retrieve
   GammaTbl    *csGammaTablePtr; // Location to copy gamma to
};
```

This routine is used after a client has used the `GetGammaInfoList` routine to iterate over the gamma tables available, 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 = 0x22)

The `cscSupportsHardwareCursor` call is used to determine whether a driver supports the hardware cursor.

```
err = Status(theDeviceRefNum, cscGetGray,
      &theVDSupportsHardwareCursorRec);
 <--     csSupportsHardwareCursor   Boolean indicating support
```

If a driver's hardware supports the Power Macintosh hardware cursor and if it implements the hardware cursor's control calls, this call should return true; otherwise, it should return false.

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 *New Macintosh Technical Notes HW-30,* described in "Apple Publications" beginning on page xxiii. 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 the MacOS is changing with the second generation of 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 `csNextResolution` 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" Monitor was sensed.
- If no display is sensed, return an error code from the Initialize routine.
- When responding to `cscGetModeTiming`, 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 `cscGetConnectionInfo`, perform the extended sense algorithm specified in the next section.
- Support DDC in the future.

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 cscGetConnectionInfo

The `cscGetConnectionInfo` call has been modified to support the new monitor sensing scheme described in the previous section. Specifically, changes has 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` 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 everytime 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 which 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 six-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 11-2 shows examples of `csConnectTaggedType` and `csConnectTaggedData` values for certain monitors.

Table 11-2 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 Display Data Channel (DDC) and would report a connection if a DDC display was 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 `cscGetTimingInfo`. 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 266.

Timing information should reflect the actual timing driving the display. For example, even if a card creates a large gDevice with hardware pan and zoom for a 13-inch RGB display it should still return `timingApple13`.

Some Apple displays (such as that for the Macintosh Quadra 840AV) support display modes such as 640x480 on a 16-inch display. The display is being driven at 16-inch timing, but the gDevice 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" beginning on page 259, 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 detailed 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.

Data Structures

MacOS 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` value is the offset to page 0 of video RAM, measured from the `BaseAddress` value returned by the `GetCurMode` routine.

```
struct VDEntryRecord {
   Ptr         csTable;        /*Pointer to color table entry*/
};

struct VDGrayRecord {
   Boolean     csMode;         /*Same as GDDevType value (0=mono, 1=color)*/
   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;        /*DisplayModeID*/
   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;    /* If the card has a direct connection
                                         to the display it returns the
                                         display component here (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 */
   UInt32          csHorizontalPixels;     /* # of pixels in a horiz line */
   UInt32          csVerticalLines;        /* # of lines in a screen */
   Fixed           csRefreshRate;          /* Vertical Refresh Rate in Hz */
   DepthMode       csMaxDepthMode;         /* Maximum Relative Bit Depth */
   UInt32          csReserved;             /* Reserved */
   UInt32          csReserved1;            /* Reserved */
};

struct VDVideoParametersInfoRec {
   DisplayModeID   csDisplayModeID;    /* ID of the resolution
                                          we want info on */
   DepthMode       csDepthMode;        /* Relative bit depth for
                                          the resolution */
   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 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;
};

typedef UInt32    DisplayModeID;
typedef UInt32    VideoDeviceType;
typedef UInt32    GammaTableID;

/* 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… force user to confirm.*/
   timingApple12       = 130,      /*  512x384 (60 Hz) Rubik timing.*/
   timingApple12x      = 135,      /*  560x384 (60 Hz) Rubik-560 timing.*/
   timingApple13       = 140,      /*  640x480 (67 Hz) HR timing.*/
   timingApple13x      = 145,      /*  640x400 (67 Hz) HR-400 timing.*/
   timingAppleVGA      = 150,      /*  640x480 (60 Hz) VGA timing.*/
   timingApple15       = 160,      /*  640x870 (75 Hz) FPD timing.*/
   timingApple15x      = 165,      /*  640x818 (75 Hz) FPD-818 timing.*/
   timingApple16       = 170,      /*  832x624 (75 Hz) GoldFish timing.*/
   timingAppleSVGA     = 180,      /*  800x600 (56 Hz) SVGA timing.*/
   timingApple1Ka      = 190,      /* 1024x768 (60 Hz) VESA 1K-60Hz timing.*/
   timingApple1Kb      = 200,      /* 1024x768 (70 Hz) VESA 1K-70Hz timing.*/
   timingApple19       = 210,      /* 1024x768 (75 Hz) Apple 19" RGB.*/
   timingApple21       = 220,      /* 1152x870 (75 Hz) Apple 21" RGB.*/
   timingAppleNTSC_ST  = 230,      /*  512x384 (60 Hz, interlaced,
                                       non-convolved).*/
   timingAppleNTSC_FF  = 232,      /*  640x480 (60 Hz, interlaced,
                                       non-convolved).*/
```

```
   timingAppleNTSC_STconv = 234,/*  512x384 (60 Hz, interlaced, convolved).*/
   timingAppleNTSC_FFconv = 236,/*  640x480 (60 Hz, interlaced, convolved).*/
   timingApplePAL_ST   = 238,      /*  640x480 (50 Hz, interlaced,
                                       non-convolved).*/
   timingApplePAL_FF   = 240,      /*  768x576 (50 Hz, interlaced,
                                       non-convolved).*/
   timingApplePAL_STconv = 242,    /*  640x480 (50 Hz, interlaced,
                                       non-convolved).*/
   timingApplePAL_FFconv = 244     /*  768x576 (50 Hz, interlaced,
                                       non-convolved).*/
};

/* 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"
                                  16" certain, 19" default, 21" certain */
   kMultiModeCRT4Connect = 7,  /* Expansion to large multi mode
                                  (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
                            (eg. 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
                                                   // cscGetNextResolution to
                                                   // reset iterator
   kDisplayModeIDNoMoreResolutions = 0xfffffffd    // Used in
                                                   // cscGetNextResolution 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
}
```

Video Services Library

The Macintosh Video Services Library (VSL) provides services for native video and graphics output drivers. At present the VSL also handles video interrupt services for vertical blanking, horizontal blanking, and other tasks. 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, including VBL interrupts.

VSLNewInterruptService

`VSLNewInterruptService` creates a new interrupt for a graphics device.

```
typedef unsigned long      InterruptServiceId;

typedef ResType            InterruptServiceType;
enum {
   kVBLService = 'vbl ';   // Vertical blanking
   kHBLService = 'hbl ';   // Horizontal blanking
   kFrameService = 'fram'; // Interlace mode
};

OSErr VSLNewInterruptService  (RegEntryIDPtr       serviceOwner,
                               InterruptServiceTypeserviceType,
                               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

The serviceOwner is the `RegEntryIDPtr` passed to the driver at install time. This is used to identify the owner. The service type is a `resType` 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` 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 where memory management calls are safe.

VSLDoInterruptService

`VSLDoInterruptService` executes tasks associated with an interrupt service.

```
OSErr VSLDoInterruptService( InterruptServiceId serviceID );
```

`serviceID` value returned by `VSLNewInterruptService`

When a graphics driver gets an interrupt, it determines which service corresponds to that interrupt and calls `VSLDoInterruptService` with the `serviceID` for that service. `VSLDoInterruptService` executes any tasks associated with the service.

VSLDisposeInterruptService

VSLDisposeInterruptService disposes of an interrupt service.

```
OSErr VSLDisposeInterruptService( InterruptServiceId serviceID );
```

`serviceID` value returned by `VSLNewInterruptService`

When a graphics driver is closing for good (the card inturrupt will no longer be serviced) it calls `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 where memory management calls are safe.

Network Drivers

This chapter describes what must be done to create STREAMS drivers for the Apple Open Transport networking hardware. 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 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. For 68K STREAMS modules, you must use the ASLM. For PowerPC modules, the CFM is the preferred mechanism 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 when linking a shared library.

IMPORTANT

Port drivers for the second generation of 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 8. ▲

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.

NativePort 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" beginning on page 73.

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 xxxxxxxx xxxxxxTD
  ```

  where `d` is the device type for Open Transport, the lower two bits state whether the driver is TPI or DLIP, and all other bits are 0.

  The macro `OTPCIServiceType(devType, isTPI, isDLPI)` should be used to create this field. The list of device types available is found in the 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`.

Experts For Port Drivers

The port scanner that Open Transport supplies registers all `.ENET`, `.TOKN`, and `.FDDI` drivers. In addition, it registers LocalTalk ports A and B, as well as the `.AIN`/`.AOUT` and `.BIN`/`.BOUT` serial drivers. If the Macintosh Communications Resource manager is present, serial drivers register as `.`*x*`IN`/`.`*x*`OUT`, where *x* is an uppercase letter from `C` to `Z`. Open Transport provides the interface modules that do the translation between the DLPI and the driver's programming interface.

Writers of any other drivers must supply a port scanner to register the driver. Port scanners are called early in the computer's startup process. Their job is to search for and register with Open Transport the port drivers they support.

The port scanner must currently be compiled as an Apple Shared Library Manager library. It will export a function call `OTScanPorts` by name from an ASLM function set with an `InterfaceID` of the constant `kOTPortScannerInterfaceID`. An example of an export file for the ASLM is shown in Listing 12-1.

Listing 12-1 Export file for an Open Transport port scanner

```
/*
   Sample .exp file for creating an ASLM
   shared library to export an Open Transport
   port scanner.
*/

#include "OpenTptModule.h"
```

```
Library
{
   /*
    * 1) Segments won't be loaded or unloaded.
    * 2) We want any libraries we depend on to be
    *    forced to load prior to us loading.
    * 3) We don't run on 68000 processors
    */

   flags = noSegUnload, forceDeps, !mc68000;

   /*
    * This id can be anything you want
    */

   id = kOTLibraryPrefix "MyScanner";

   /*
    * Put the appropriate version number here.
    */

   version = 1.0a1;
   memory = client;
};

FunctionSet MyScannerFunction
{
   /*
    * You must use this InterfaceID
    */

   InterfaceID= kOTScannerInterfaceID;

   /*
    * Your ID can be anything as long as it starts
    * with kOTScannerPrefix.
    */

   id    = kOTScannerPrefix "MyPScnrName,1.0";

   /*
    * You can export other functions, but you must
    * have "extern OTScanPorts" somewhere in the list.
    */

   exports  = extern OTScanPorts;
};
```

The `OTScanPorts` function is a C function with the following prototype:

```
void OTScanPorts(void);
```

When the `ScanPorts` function is called, the scanner should search for the hardware and drivers that it supports and call the `OTRegisterPort` function to register each hardware driver.

```
OSErr OTRegisterPort (const char* driverName,
                      const char* logicalName,
                      const char* portName,
                      OTPortRef ref, UInt32 flags,
                      void* cookie);
```

Field descriptions

`driverName` The name of the driver (the value in the `streamtab` and the name used to find your driver entry points). This name must be unique among all STREAMS drivers. The user does not normally see this name; it is used only to find and load the STREAMS driver.

`logicalName` The logical name of the module, the name that users see. It is typically related to the hardware type of the driver, and all drivers that control the same hardware type usually have the same name. Clients of Open Transport can use its location-dependent syntax for devices to make driver specified by `logicalName` unique.

`portName` This is a unique name for the particular hardware device that the driver will control. A single driver may be registered multiple times—once for each hardware device that the driver can control. The `driverName` and `logicalName` values will be the same for each of these registrations, but the `portName` must be unique. A common technique is to create the `portName` by appending successive digits to the `driverName`. If you want to create the `portName` from the `logicalName`, you must do so by appending the two-digit slot number of the device. By following this rule you always create unique port names.

`ref` This is a port reference value, which must also be unique within a computer. This value must be created by calling

`OTCreateNuBusPortRef(`*`dev`*`, `*`slot`*`, `*`other`*`)` or
`OTCreatePCIPortRef(`*`dev`*`, `*`slot`*`, `*`other`*`)`

where *`dev`* is one of the Apple-supplied device types defined in the file `OpenTptLinks.h`, *`slot`* is the slot number that the device corresponds to, and *`other`* is a differentiating number (for instance, the port number on a multi-port serial card). A slot number of 0 is used for the main logic board.

`flags` `Flags` is a bitmap that is set to one or more of the following values, combined by logical OR:
`kOTModuleIsDLPI` = 0x00000001
`kOTModuleIsTPI` = 0x00000002
`kOTPortIsDefault` = 0x80000000

These values describe the interface to the module, and specify whether or not the module is the default port for all modules with the specified `logicalName`.

`framingFlags` `framingFlags` is a bitmap that is set to 0 if the device only supports a single type of framing or to a bitmap indicating which framing options the port supports. If you do not use 0 for `framingFlags`, you must support the `kOTSetFramingType IOCTL` call, which will pass you a 32-bit value with a single bit set, indicating the framing option desired. If this `IOCTL` call is made, your DLPI driver should fill in the `dl_mac_type` field of a `dl_info_ack_t` with a value consistent with the requested framing type.

For example, Ethernet supports four framing options:

`kOTFramingEthernet`	= 0x01
`kOTFramingEthernetIPX`	= 0x02
`kOTFraming8023`	= 0x04
`kOTFraming8022`	= 0x08

Most Ethernet drivers support all framing options except `kOTFraming8023`. Typically, Ethernet drivers support `kOTFraming8022`, which indicates that they can handle full demultiplexing for service access points and subnet access protocols, whereas `kOTFraming8023` indicates that they will deliver all 802.3 frames to a single client. If a client requests `kOTFraming8022` using the `IOCTL` call, then the driver should return `DL_CSMACD` in the `dl_mac_type` field. If a client requests `kOTFramingEthernet` or `kOTFramingEthernetIPX`, then the driver should return `DL_ETHER` instead.

Currently, Open Transport does not support the `kOTFraming8023` framing type, so Ethernet drivers must handle test/XID frames and perform full demultiplexing for service access points and subnet access protocols to work properly with AppleTalk and TCP/IP.

`cookie` This parameter is any value you wish to supply to your driver for this particular instance. It can be as simple as the slot number of the device, or as complex as a pointer to a data structure. The driver can use Open Transport functions to retrieve this cookie based on the major number given to the driver by the STREAMS environment. See “Miscellaneous Services” beginning on page 288, for further information about using the `cookie` parameter.

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.

If the `ValidateHardware` function is exported, then one of the following values should be returned:

`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 not OK. The port will not be registered, and the driver will be unloaded from memory.
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;
   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`	Your driver can be entered once from the upper queue and once from the lower queue at the same time.
`SQLVL_QUEUEPAIR`	Your driver can be entered from either the upper queue or the lower queue, but not at the same time.
`SQLVL_MODULE`	Your driver can only be entered once, no matter which instance of the module is entered (that is, if your driver is handling multiple ports, it will only be entered by one port at a time).
`SQLVL_GLOBAL`	Only one STREAMS module or driver can be entered at any one time. Between 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_count` This field is used by Open Transport to keep track of when a driver was first loaded and when it was last unloaded. It should be initialized to 0.

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);
```

This function must call `OTInitModule()` as the first thing that it does.

```
Boolean OTInitModule (void);
```

If the `OTInitModule` function returns `false`, the `InitStreamModule` function should immediately return `false` and do nothing else.

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 PCI drivers, it's value is `RegEntryPtr`.

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);
```

This function must call `OTTerminateModule()` as the last thing that it does.

```
void OTTerminateModule (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. Either way, a driver must call `OTInitModule` and `OTTerminateModule` either from the Open Transport initialization and termination routines or from the DLL initialization and termination routines.

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 b e most compatible with Mac OS. 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.

Secondary Interrupt Services

The functions described in this section are associated with Open Transport's secondary interrupt services.

```
typedef void (*OTDeferredProcPtr)(void* contextInfo);
```

This typedef defines the deferred task callback function.

```
long OTCreateDeferredTask  (OTDeferredProcPtr   proc,
                            long                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.

```
void mi_timer(mblk_t* timerMsg, unsigned long milliSeconds);
```

This function schedules the `timerMsg` (which must be created using `mi_timer_alloc`) to be placed on the target STREAMS queue at a specified future time.

```
void mi_timer_cancel(mbk_t* timerMsg)
```

This function cancels an outstanding timer message. The `timerMsg` 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_q_switch (mblk_t* timerMsg, queue_t* newTarget)
```

It may become necessary to target a scheduled timer message to another queue by using the following function:

```
void mi_timer_free(mblk_t* timerMsg)
```

This function cancels and frees the specified timer message; `mi_timer_cancel` does not free the 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 cancelled. In this case, the correct course of action is to ignore the message (that is, don't free it).

Atomic Services

Open Transport supplies atomic services that help reduce the need for drivers to disable and enable interrupts.

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)
```

The second set of services atomically increments and decrements a long variable. The parameter is a pointer to the long, and the return value is the new value of the long.

```
long OTIncrementLong(long* longToIncrement)
long OTDecrementLong(long* longToDecrement)
```

The third service is a general compare and swap. It determines whether the pointer at `thePlace` still contains `oldValue`, and if so, it substiututes `newValue`. If the compare and swap succeeds, the function returns `true`, otherwise `false`.

```
Boolean OTCompareAndSwap
   (void* oldVal, void* newVal, void** thePlace)
```

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 of us 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. 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.

```
void* OTEnqueue
   (void** list, void* newListHead, size_t offsetOfNextPtr);
void* OTDequeue(void** theList, size_t offsetOfNextPtr);
```

Power Services

PCI network devices that can change their power levels should conform to these rules:

- A call to `ValidateHardware` sets the device to either low power or no power, as appropriate for the device.
- A call to `InitStreamModule` sets the device to its normal power level.
- A call to `TerminateStreamModule` sets the device to either low power or no power, as appropriate for the device.

In addition, network devices that can change their power levels should support the `kOTSetPowerLevel` `IOCTL` call.

Following are the four-byte selectors that can be passed to the `IOCTL` call, with their return values:

`'psup'`	Return a value of 1 if the card supports power control, 0 if it does not.
`'ptog'`	Return a value of 1 if the card supports switch between high and low power after initialization, 0 if it does not.
`'psta'`	Return a value of 1 if the card is in high power mode
`'pmx5'`	Returns the card's maximum power consumption in microwatts from the 5 V supply while in high-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
`'pmn3'`	Returns the card's maximum power consumption in microwatts from the 3.3 V supply while in low power mode.
`'splo'`	Sets the card to low power mode. Returns a value of 0 if completed successfully, `OSErr` if not.
`'sphi'`	Sets the card to high power mode. Returns a value of 0 if completed successfully, `OSErr` if not.

Miscellaneous Services

```
TPortRecord* OTFindPortByDev (dev_t dev, const char* logName)
```

The `OTFindPortByDev` function can be used in your module's `open` routine to obtain the `TPortRecord` related to your port. From this record, you can retrieve the cookie that the port scanner saved for the driver, as well as other useful information (see "Experts For Port Drivers" beginning on page 278).

```
struct TPortRecord
{
   void*                fLink;
   void*                fSelf;
   char                 fPortName[kMaxProviderNameSize];
   struct ModuleRecord* fModule
   struct TPortRecord*  fAlias;
   UInt32               fFramingFlags;
   OTPortRef            fRef;
   void*                fContext;
   void*                fExtra;
};

struct ModuleRecord
{
   void*                fLink;
   void*                fSelf;
   UInt32               fFlags;
   UInt16               fType;
   char                 fLogicalName[kMaxModuleNameSize];
   char                 fModuleName[1];
};
```

The `OTFindPortByDev` function returns the actual `TPortRecord` used by the system, so don't modify it. The only really useful fields are the `fRef` field, which contains the `OTPortRef` for your module, and the `fContext` field, which contains the `cookie` that the port scanner saved for the driver. For systems using the native driver architecture, `cookie` is a pointer to a structure whose first element is the `RegEntryID` for the driver; in other words, `cookie` can be interpreted as a `RegEntryIDPtr`.

The `OTFindPortByDev` function is most useful for drivers that handle multiple hardware devices. By using the `fRef` or `fContext` value, the driver can determine which hardware device the open call is referring to.

Ethernet Driver

The Open Transport Ethernet driver is a STREAMS driver that presents a Data Link Provider Interface (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 Ethernet-style 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 Ethernet driver. The ones marked with a † are not required by either the Appletalk or TCP/IP stacks:

DL_INFO_REQ
DL_INFO_ACK
DL_BIND_REQ
DL_BIND_ACK
DL_UNBIND_REQ
DL_SUBS_BIND_REQ
DL_SUBS_BIND_ACK
DL_ENABLEMULTI_REQ
DL_DISABLEMULTI_REQ
DL_OK_ACK
DL_ERROR_ACK
DL_UNITDATA_REQ
DL_UNITDATA_IND
DL_TEST_REQ †
DL_TEST_IND †
DL_TEST_RES †
DL_TEST_CON †
DL_XID_REQ †
DL_XID_IND †
DL_XID_RES †
DL_XID_CON †
DL_PHYS_ADDR_REQ
DL_PHYS_ADDR_ACK

Future versions of the driver will also support the additional primitives
DL_GET_STATISTICS_REQ †
DL_GET_STATISTICS_ACK †
DL_SET_PHYS_ADDR_REQ

Extensions to the DLPI

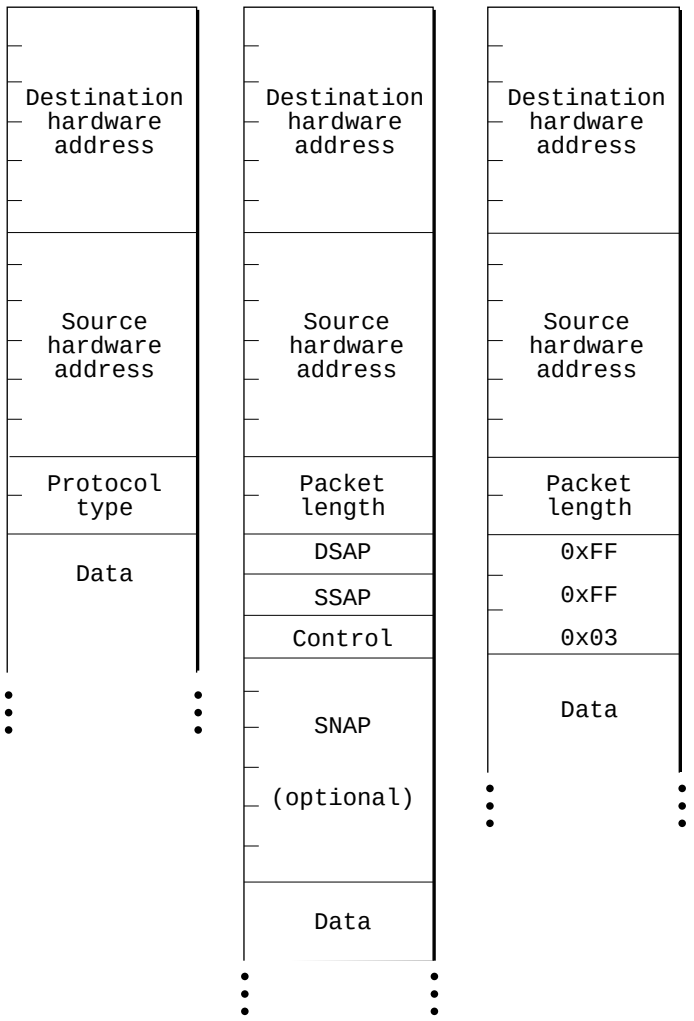
In addition to supporting the DLPI primitives listed above, the Open Transport Ethernet driver includes extensions to support Mentat's Fast Path mode (described in "Fast Path Mode" on page 295). This includes the handling of M_IOCTL messages with a type of DL_IOC_HDR_INFO and special handling of M_DATA messages.

Packet Formats

The Open Transport Ethernet driver recognizes three packet formats. They are Classic Ethernet (using DIX), 802.2, and IPX. The details of the packet format are largely hidden from the client by the driver.

Packet formats are diagrammed in Figure 12-1.

Figure 12-1 Packet formats



Note
The 802.2 standard is described in *Logical Link Control,* ANSI/IEEE Standard 802.2-1985. ◆

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, 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.

Classic Ethernet Packets

In Classic 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 ($5DD to $FFFF).

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

The protocol-dependent section of IPX packets is 5 bytes long, consisting of a 2-byte length field followed by 2 bytes of $FF and one byte of $03.

Note
It is possible to differentiate 802.2 packets and IPX packets because a valid 802.2 packet will never have both the DSAP and SSAP fields equal to $FF. ◆

Address Formats

Addresses used by the Open Transport Ethernet 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 ones is the broadcast address. If a hardware address is not all ones but the low bit of the first (leftmost) 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.

Classic Ethernet

In Classic Ethernet, the DLSAP corresponds to the protocol type field.

802.2

In 802.2 packets, the DLSAP corresponds to the SSAP (in a `DL_BIND_REQ`, `DL_BIND_ACK`, or in the source address field of a `DL_UNITDATA_IND`), or the DSAP (in a `DL_UNITDATA_REQ` or in the destination address field of a `DL_UNITDATA_IND`). If the DLSAP is $AA, then it must be followed by a 5-byte SNAP.

IPX

In IPX packets, the DLSAP is always $00FF.

Binding

The information passed in a Bind Request is a function of the type of packets to be handled by this stream—classic 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 that 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 not supported by the Open Transport Ethernet Driver.

Classic 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 ($5DD to $FFFF). The `dl_xidtst_flg` field is ignored. On any given hardware port, only one stream at a time can be bound to a specific Ethernet protocol.

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. ($0 to $FE). 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 $AA, 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`.

Note
Attempting to perform a hierarchical `subs_bind` operation to any SAP value other than $AA will cause an error. ◆

On any given hardware port, only one stream at a time may be bound to a specific 802.2 protocol.

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 ($1 to $FD). In this case, the `dl_subs_bind_class` field should be set to `DL_PEER_BIND`. Note that SAP 255 ($FF) is the global (broadcast) SAP, and is always enabled.

Note
For a description of group and global SAPs, see ANSI/IEEE Standard 802.2-1985. ◆

Note
When sending packets to DLSAP $FF, 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 ($FF). The `dl_xidtst_flg` field is ignored. On any given hardware port, only one stream at a time may be bound to the IPX protocol.

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" beginning on page 292.

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 an stream bound to the IPX SAP can send to another IPX stream.

Note
The data portion of an IPX packet must always begin with the 3-byte IPX header $FFFF03. ◆

To support Mentat's Fast Path mode, the Open Transport Ethernet driver treats `M_DATA` messages as fully formed 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" on page 295 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).

With the exception of IPX, the data portion of the message consists of everything following the protocol–dependent section. For IPX packets, the data portion of the message starts with the 3-byte IPX header $FFFF03.

Test and XID Packets

An Ethernet driver should include 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 may not 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 pre-computed 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, Ethernet 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 be DL_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.

TokenRing and FDDI Drivers

Open Transport TokenRing and Fiber Distributed Data Interface (FDDI) drivers are identical to the Ethernet Driver with only 802.2 packets and addressing supported. A hardware multicast is a hardware address with the two high order bits of the leftmost byte set to 1.

SCSI Drivers

This chapter discusses the requirements for writing native driver code to support SCSI devices on PCI cards in the second generation of 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-indendence 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" on page xxiii, 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 the second generation of Power Macintosh computers. For a discussion of experts, see "Terminology" beginning on page 47.

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 SCSI Interface Module (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 layer (XPT) 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 will prepare the code fragment and call 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` will cause the code fragment to be unloaded. Note that this is a `ProcPtr` based interface so you must pass `UniversalProcPtr` structures for all entrypoints. 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 13-1.

Listing 13-1 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 categorys
      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 a SIM from other SIMs and PCI cards. For NuBus the slot number and `sRsrcID` were sufficient, but these values don't exist for PCI. Instead, a 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 these fields and supply the 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 for this purpose. 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;    // <-
UInt8                simSRsrcID;       // <-
RegEntryIDPtr        simRegEntry;      // ->
```

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` from native code to call the XPT's entry points. This approach was outlined earlier in "SIMs for Current Versions of Mac OS" beginning on page 298. Universal procedure pointers are described in *Inside Macintosh: PowerPC System Software,* listed in "Apple Publications" on page xxiii.

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 7, "Writing Native Drivers." Primarily, you should limit your communication with Mac OS to the calls documented in Chapter 9, "Driver Services Library."

The DAV Interface

This part of *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the hardware and software characteristics of the Macintosh digital audio/video (DAV) interface in the second generation of Power Macintosh computers. The DAV feature lets expansion cards access the computer's digital sound and video data streams independently of the PCI local bus. This part consists of the following chapters:

- Chapter 14, "Accessing AV Data," describes the general mechanical and electrical characteristics of the DAV interface and summarizes the system software that Apple supplies for DAV signal management.
- Chapter 15, "Audio Hardware Interface," provides electrical and timing details of the digital sound I/O signals that are present at the DAV interface.
- Chapter 16, "Audio Programming," describes the Macintosh system programming interface (SPI) that lets expansion cards, drivers, and applications control the sound signals at the DAV interface.
- Chapter 17, "Video Hardware Interface," provides electrical and timing details of the video I/O signals at the DAV interface.
- Chapter 18, "Video Programming," describes the Macintosh SPI that lets expansion cards, drivers, and applications control the video signals at the DAV interface.

PART FOUR

Accessing AV Data

The DAV interface in the second generation of Power Macintosh computers is supported by a single internal connector that is separate from the computer's PCI connectors. Expansion cards can be designed to plug into both the PCI bus and the DAV connector. The user can install one such card at a time in the computer.

The ***DAV connector*** taps into the Power Macintosh system's video and sound data streams, providing access to the system's 4:2:2 unscaled digital video input signal and the digital audio signal input for the system's sound encoder/decoder (codec). An expansion card can capture or generate these signals without having to pass them through the PCI bus.

The DAV interface gives expansion cards greater speed and facility in processing video and sound data, because cards can access data and perform PCI bus transactions independently. For example, the DAV interface supports high-performance hardware audio or video compression and decompression capabilities on expansion cards. A card can access raw data through the DAV interface and can transfer compressed data over the PCI bus to and from system memory or disk storage. Because the card accesses raw and compressed data through two separate interfaces, it can achieve high processing rates.

Apple provided a different DAV interface in certain models of the first generation of Power Macintosh computers, which used NuBus. This previous interface is described in *Macintosh Developer Note Number 8*, listed in "Apple Publications" beginning on page xxiii. The new PCI-based DAV interface documented here has a different hardware configuration and includes new system software controls. The hardware configuration is described in this chapter; the software controls are described in Chapter 16, "Audio Programming," and Chapter 18, "Video Programming."

Mechanical Implementation

The NuBus-based first generation of Power Macintosh computers put the video part of the AV circuitry on an optional plug-in card called the *AV card.* In the PCI-based second generation of Power Macintosh computers, all circuitry that supports video I/O operations is located on the main logic board. There is no optional AV card.

In AV-equipped second-generation Power Macintosh models, audio and video signals from the main circuit board are fed to the DAV connector, which is mounted near one PCI expansion card connector. To take advantage of the DAV interface, a PCI expansion card must be designed with a 60-conductor flat ribbon cable that plugs into the DAV connector.

Interface Connector

The DAV connector in the second generation of Power Macintosh computers is a 60-pin dual-row type with 0.100 inch pin spacing. Its pin assignments are shown in Table 14-1.

Table 14-1 DAV connector pin assignments

Pin	Signal	Pin	Signal	Pin	Signal
1	Ground	21	Y bit 5	41	Ground
2	Reserved	22	Y bit 4	42	Analog audio input left
3	Ground	23	Y bit 3	43	Analog audio input common
4	Reserved	24	Y bit 2	44	Analog audio input right
5	Ground	25	Y bit 1	45	Ground
6	Reserved	26	Y bit 0	46	Digital audio input
7	Ground	27	Ground	47	Ground
8	Reserved	28	LLClk*	48	Digital audio output
9	Ground	29	Ground	49	Ground
10	Reserved	30	CREFB†	50	Digital audio clock
11	UV bit 7	31	Ground	51	Ground
12	UV bit 6	32	Vertical sync	52	Digital audio sync
13	UV bit 5	33	Ground	53	Ground
14	UV bit 4	34	Reserved	54	SVidInC
15	UV bit 3	35	Ground	55	VidInGnd
16	UV bit 2	36	HRef	56	SVidInY
17	UV bit 1	37	Ground	57	VidInGnd
18	UV bit 0	38	DIR‡	58	Reserved
19	Y bit 7	39	IIC data§	59	Reserved
20	Y bit 6	40	IIC clock§	60	Reserved

* Line-locked clock
† Clock reference qualifier.
‡ Expansion bus input, pulled down by 1 kΩ.
§ Inter-IC control signals, a Philips standard.

Using the DAV Interface

This section briefly outlines two of the many possible scenarios for using the DAV interface in the design of PCI expansion cards for the second generation of Power Macintosh computers.

Video Compression Hardware

Developers can provide compression/decompression hardware on a Macintosh PCI expansion card to improve the performance of QuickTime movie captures. This scenario can be further divided into two situations:

- Movie capture: when capturing a movie, the expansion card uses hardware to compress the incoming video.
- Movie playback using hardware decompression: the expansion card hardware is used for playback. An example is an MPEG decoder.

Teleconferencing

The DAV hardware can be used for teleconferencing. The requirements for such a card are more complex than those of a video compression card, because a teleconferencing card may have additional sound and display requirements.

Audio Hardware Interface

Macintosh computers whose model designations end in AV support high-quality 16-bit stereo sound processing. This capability lets expansion cards capture and generate digital sound data suitable for professional broadcasting uses and commercial CD-ROM recording. It also lets software support advanced audio features such as speech recognition and natural speech synthesis. The DAV interface in the AV computers lets expansion cards access the computer's digital audio data stream.

This chapter helps you understand the audio portion of the DAV interface by providing details of the audio signal-processing features in the second generation of Power Macintosh computers. The first section describes the user's audio interface to external equipment. The next section, "Sound Processing," provides technical information about the Apple chips that support audio signal digitizing and amplification. The last section, "Sound Frames," discusses the timing of data frames in the flow of digital audio signals.

External Audio Signals

All Power Macintosh computers contain external stereo mini phone jacks for sound I/O, connected through amplifiers to a sound codec (coder/decoder) chip. In the second generation of Power Macintosh computers, the codec is called the ***audio waveform amplifier and converter (AWAC)*** chip. The sound system achieves simultaneous 16-bit broadcast-quality stereo sound input and output, with buffering, and supports Apple's speech synthesis and recognition software.

Table 15-1 describes the external sound I/O signals.

Table 15-1 Power Macintosh sound signals

Panel label	Description
Audio In	8 kΩ impedance, 2 V rms maximum, 22.5 dB gain available
Audio Line Out	37 Ω impedance, 0.9 V rms maximum, attenuated –22.5 dB (crosstalk degrades from –80 dB to –32 dB when the audio output is connected to 32 Ω headphones)

Sound I/O bandwidth is 20 Hz to 20 kHz, plus or minus 2 dB. Harmonic distortion and noise total less than 0.05 percent over the bandwidth with a 1 V rms sine wave input. The input signal-to-noise ratio (SNR) is 82 dB, and the output SNR is 85 dB with no audible discrete tones.

All Macintosh computers are supplied with a built-in speaker. Software can control the volume of sound to the built-in speaker and to the sound output connector independently. Apple also offers a compatible high-quality microphone for the AV computers that is specifically designed for speech recognition applications.

Sound Processing

The AWAC sound codec chip conforms to the IT&T *ASCO 2300 Audio-Stereo Codec Specification* (listed in "Other Publications" beginning on page xxv) and furnishes high-quality 16-bit stereo sound I/O. It uses time-division multiplexing to transfer multiple audio channels between the DAV connector and the Macintosh system for direct memory access (DMA) transfers to and from RAM memory. The sound signals that appear at the Power Macintosh DAV connector are listed in Table 15-2.

Table 15-2 DAV interface sound signals

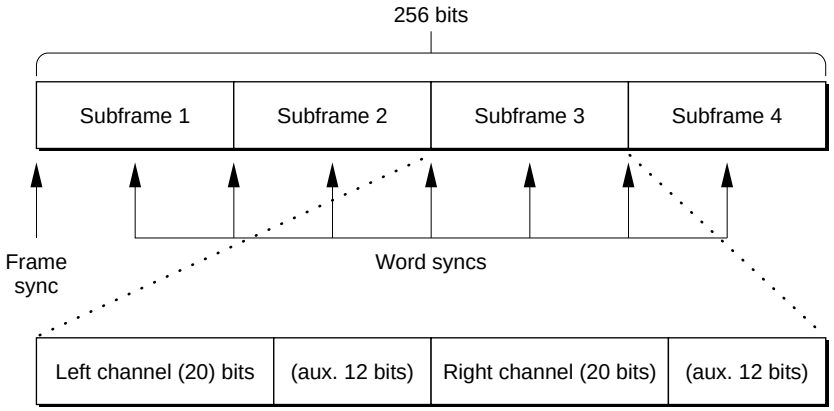
DAV pin	Signal	Description
46	Digital audio in (AwacDataIn)	Sound input from DAV connector to system
48	Digital audio out (AwacDataOut)	Sound output from system to DAV connector
50	Digital audio clock (AwacClk)	Bit clock that clocks serial data on digital audio out and in; 256 times the sample rate; also used to clock digital audio sync
52	Digital audio sync (AwacSync)	Signal that marks the beginning of a frame and a word

DAV sound signals have a minimum setup time of 10 ns and a minimum hold time of 8 ns; they can tolerate a maximum load of 20 pF.

Sound Frames

The AWAC sound codec in the Power Macintosh hardware transfers data in 256-bit ***frames,*** each of which contains four ***subframes*** of 64 bits each. Each subframe carries two 32-bit audio samples, one for the left stereo channel and one for the right stereo channel. Each sample contains 20 data bits and 12 auxiliary bits. Subframe 1 is reserved for the Macintosh system sound I/O; the other subframes are available for applications and expansion cards to use. The audio frame structure is shown in Figure 15-1.

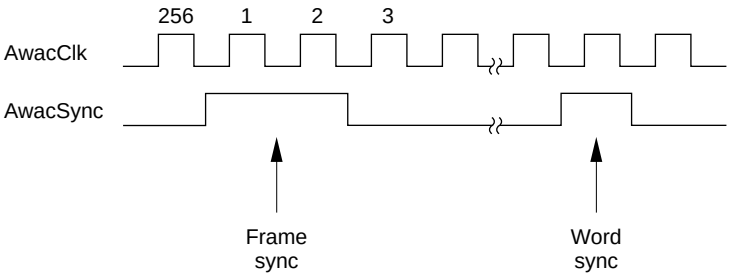
Figure 15-1 AWAC sound frame



The signals AwacSync, AwacDataOut, and AwacDataIn are clocked by the AwacClk signal. The falling edge of the clock is used to activate the signals, and the rising edge is used to sample them. The AwacClk frequency may be 6.144, 8.192, or 12.288 MHz.

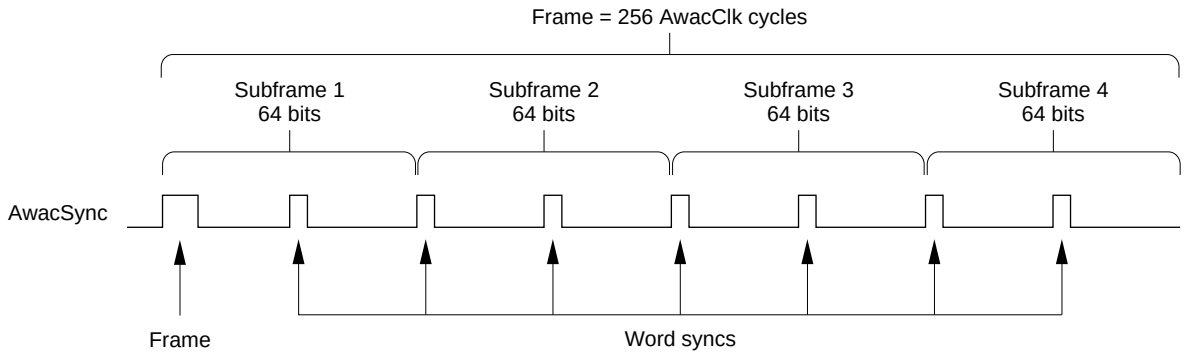
As shown in Figure 15-2, a frame sync is marked by a pulse two AwacClk cycles wide; a word sync is marked by a pulse one AwacSync cycle wide.

Figure 15-2 Sound frame and word synchronization



The AwacSync synchronization signals for each subframe are shown in Figure 15-3.

Figure 15-3 Sound subframe synchronization



Audio Programming

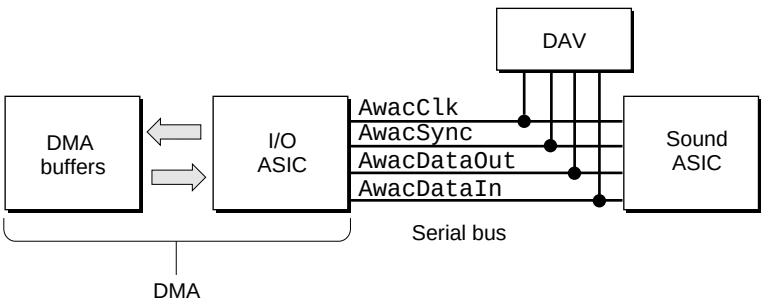
This chapter describes the programming interface that Apple provides for controlling DAV audio signals in PCI-based Power Macintosh computers. This programming interface provides a flexible way for expansion card developers to configure the audio portion of the computer's DAV interface to meet the requirements of their cards. It also supports the development of third party input drivers and output sifters that handle DAV audio data.

Audio Subframes

The audio portion of the DAV interface is a serial data bus with four time-division multiplexed stereo channels of audio. The four channels are called subframes and have separate input and output serial lines. Currently, all PCI-based Power Macintosh models with the DAV interface use only one of the four subframes—subframe 0—for audio I/O. Multiple channels of audio can be supported by enabling more than one subframe. In all cases, the audio for the enabled subframes is interleaved in the DMA audio buffers.

Figure 16-1 shows a simplified physical layout of the typical audio data flow in a Power Macintosh computer.

Figure 16-1 Audio data flow



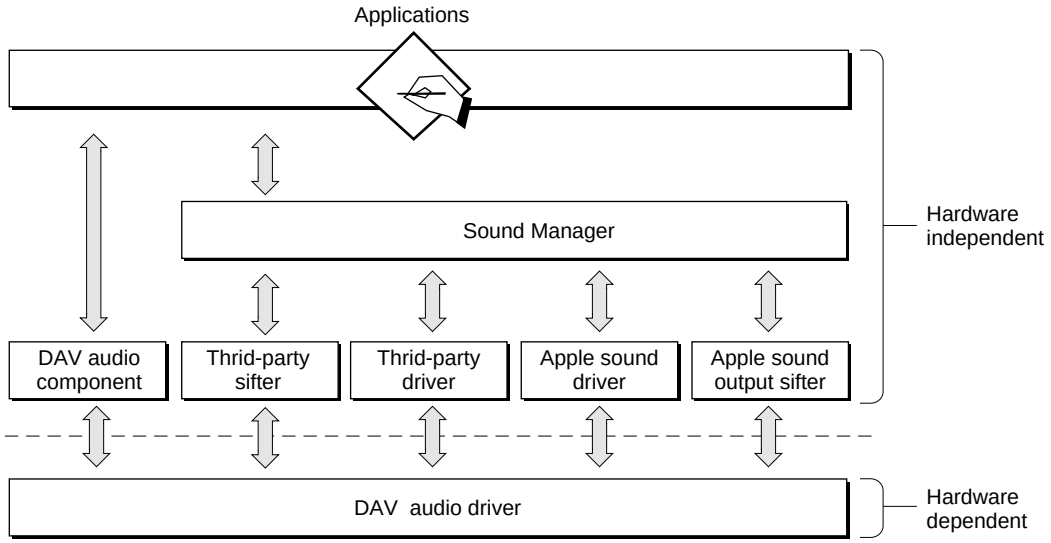
In Figure 16-1, the signals AwacDataIn and AwacDataOut are the serial input and output lines; AwacClk and AwacSync are the clock and synchronization lines. For more information about DAV audio signals, see Chapter 15, "Audio Hardware Interface."

Audio Software Architecture

The architecture of the DAV audio programming interface isolates all hardware-specific code—including DAV software, the DMA implementation, and codec-specific code—in a separate ***DAV audio driver,*** which is written in native PowerPC code. The DAV audio driver is implemented as a PowerPC native driver to provide maximum performance and the desired functionality with minimum system overhead.

The relations between the DAV audio driver, the Macintosh Sound Manager, and various Apple and third-party audio software components is diagrammed in Figure 16-2. For more information about the Sound Manager, see *Inside Macintosh: Sound.* This book is listed in "Apple Publications" beginning on page xxiii. The DAV audio components shown in Figure 16-2 is described in the next section.

Figure 16-2 Audio software architecture



DAV Audio Component

The DAV ***audio component*** is an audio configuration utility that is part of the Macintosh system software in the second generation of Power Macintosh computers. It is used by PCI expansion cards that must make configuration changes to the DAV audio subsystem but do not need to provide their own sound input driver and output sifter. Examples include a video teleconferencing card that wants to play audio through the computer's built-in sound system and a multichannel DAV audio card that has its own audio engine.

The DAV audio component has an external interface consisting of two calls:

```
DAVAudioSetInfo   (ComponentInstance ti,
                   OSType selector,
                   void *infoPtr)

DAVAudioGetInfo   (ComponentInstance ti,
                   OSType selector,
                   void *infoPtr)
```

Drivers, expansion card firmware, and applications can use these calls to configure the DAV audio subsystem or to get information on the current DAV audio configuration. The type of operation is determined by selector. The valid selector values are

```
enum {
   diActiveChannels     = 'chan',
   diPlayThrough        = 'plth',
   diInstallIntProc     = 'inst',
   diRemoveIntProc      = 'remv',
   diActive             = 'actv',
   diAuxBits            = 'aux '
}
```

The selected operations are described in the next sections.

diActiveChannels

When `diActiveChannels` is selected, the `DAVAudioSetInfo` call configures the DAV subsystem to enable the subframe specified by `infoPtr`. When more than one subframe is enabled, the audio data for the various subframes are interleaved in the buffer (see "diInstallIntProc" on page 317).

The `DAVAudioGetInfo` call returns the currently active subframes.

For the `DAVAudioSetInfo` call, the `infoPtr` parameter is a `DAVSubFrame` type; you add together all the desired subframes. For the `DAVAudioGetInfo` call, `infoPtr` is a pointer to a `DAVSubFrame`. This type may have any combination of the following values:

```
typedef UInt32 DAVSubFrame;
enum {
   kInputSubFrame0   = 0x0001,   // Available input subframes
   kInputSubFrame1   = 0x0002,
   kInputSubFrame2   = 0x0004,
   kInputSubFrame3   = 0x0008,

   kOutputSubFrame0  = 0x1000,   // Available output subframes
   kOutputSubFrame1  = 0x2000,
   kOutputSubFrame2  = 0x4000,
   kOutputSubFrame3  = 0x8000
};
```

diPlayThrough

When `diPlayThrough` is selected, the `DAVAudioSetInfo` call configures the DAV subsystem to enable playthrough on the subframe specified by `infoPtr`. Playthrough mixes the input from the enabled subframe with the system output, which is subframe 0. For example, a card may want to play data into the DAV subsystem and have it routed to

the Macintosh built-in speaker. To do this, it drives one of the available input subframes (1 through 3) and enables playthrough for that subframe.

The `DAVAudioGetInfo` call returns the subframes currently enabled for playthrough. The only valid subframes enabled for playthrough are input subframes 1 through 3.

For the `DAVAudioSetInfo` call, the `infoPtr` parameter is a `DAVSubFrame` type; you add together all the desired subframes. For the `DAVAudioGetInfo` call, `infoPtr` is a pointer to a `DAVSubFrame`. This type may have one of the values listed in the previous section.

diInstallIntProc

When `diInstallIntProc` is selected, the `DAVAudioSetInfo` call installs an audio interrupt service routine. When the audio hardware requires more data for output or has filled its buffer during input, this routine will be called to service the buffer. The format of the callback is as follows:

```
typedef pascal void (*DAVAudioProcPtr) (short         *buff,
                                        UInt32        numBytes,
                                        DAVSubFrame   format,
                                        long          refCon);

enum {
   uppDAVAudioProcInfo = kPascalStackBased
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short *)))
      | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(UInt32)))
      | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(DAVSubFrame)))
      | STACK_ROUTINE_PARAMETER(4, SIZE_CODE(sizeof(long)))
};
```

The callback routine will be passed a pointer to the data in `buff`, the number of bytes in the buffer in `numBytes`, the format of the buffer in `format` and a client-installed reference constant in `refCon`. The format specification is a bit map of the active DAV subframes. For example, if the buffer contains samples from both subframe 0 and subframe 1, `format` will contain `0x3 (kDAVSubframe0 + kDAVSubframe1)`. Because the audio samples from the various subframes are interleaved in the buffer, interrupt handlers should use this information to parse the buffer as necessary.

The `infoPtr` parameter points to a `DAVProcStructure`:

```
typedef struct {
   DAVAudioProcPtr      proc;
   long                 refCon;
   DAVIOSpec            io;
} DAVProcStructure;
```

The `DAVIOSpec` field can have these values:

```
typedef UInt32 DAVIOSpec;
enum {
   kInput      = 1,
   kOutput     = 2
};
```

There is no `DAVAudioGetInfo` call for the `diInstallIntProc` selector.

diRemoveIntProc

When `diRemoveIntProc` is selected, the `DAVAudioSetInfo` call removes a previously installed audio interrupt service routine.

The `infoPtr` parameter points to a `DAVProcStructure`, described in the previous section.

There is no `DAVAudioGetInfo` call for the `diRemoveIntProc` selector.

diActive

When `diActive` is selected, the `DAVAudioSetInfo` call activates or deactivates the Macintosh DAV audio engine. The `DAVAudioGetInfo` call for the `diActive` selector returns the current status of the audio engine.

The `infoPtr` parameter contains a `DAVDMAStatus` value:

```
typedef UInt32 DAVDMAStatus;
enum {
   kDMAInputActive   = 0x0001,   // Set to activate input
   kDMAOutputActive  = 0x0002,   // Set to activate output
   kDMABothActive    = 0x0003,   // Set both bits to activate both
   kDMAInputInactive    = 0x0004,   // Set to deactivate input
   kDMAOutputInactive   = 0x0008,   // Set to deactivate output
   kDMABothInactive     = 0x000C    // Set both to deactivate both
};
```

diAuxBits

The output auxiliary bits in the DAV audio subframe can transmit configuration and control information to an expansion card connected to the DAV interface, using `diAuxBits`. The input auxiliary bits can be used to read status from such a card.

The `infoPtr` parameter points to a `DAVAuxStructure`:

```
typedef struct {
   UInt32         bits;
   DAVSubFrame    subframe;
} DAVAuxStructure;
```

When `diAuxBits` is selected, the `DAVAudioSetInfo` call sends the specified auxiliary bits to the subframe specified by the `subframe` field of `DAVAuxStructure`. Note that the specified subframe must first be enabled by calling `DAVAudioSetInfo` with the `diActiveChannel` selector. The `DAVAudioGetInfo` call for the `diAuxBits` selector returns the auxiliary bits for the specified subframe.

Macintosh Audio Driver Interface

The Macintosh DAV audio driver is the lowest layer of audio software, the layer that communicates with the hardware. Calls are made directly to the driver from sound input drivers and output sifters.

Note
The information in this section is relevant only if you must provide an input driver or output sifter for a PCI expansion card connected to the DAV interface. An example would be a stereo digital I/O card. ◆

The DAV audio driver programming interface consists of seven control and status calls accessible by third parties. The driver has been partitioned into three sections:

- DAV-specific
- DMA-specific
- Codec-specific

All of the DAV-specific and DMA-specific and one of the codec-specific calls are accessible to third party software. This lets you make a custom input driver or output sifter for your I/O expansion card. The DAV audio driver interface call selectors are listed in Table 16-1.

Table 16-1 Audio driver interface call selectors

Selector	Call types
DAV-specific:	
`davSystemSubframe`	control and status
`davActiveChannels`	control and status
`davPlayThrough`	control and status
DMA-specific:	
`dmaInstallIntProc`	control
`dmaRemoveIntProc`	control
`dmaActive`	control and status
Codec-specific:	
`cdcAuxBits`	control and status

You make calls to the DAV audio driver through the `DAVCtlParam` data structure shown below. The `ioCRefNum` field contains the driver's `refNum`, the `csCode` field contains the value 2, the `csParam[0]` field contains the selector shown in Table 16-1, and the `csParam[1-4]` field contains any necessary call parameters. All other fields in the `DAVCtlParam` data structure should contain 0.

```
struct DAVCtlParam {
   QElemPtr          qLink;
   short             qType;
   short             ioTrap;
   Ptr               ioCmdAddr;
   IOCompletionUPP   ioCompletion;
   OSErr             ioResult;
   StringPtr         ioNamePtr;
   short             ioVRefNum;
   short             ioCRefNum;
   short             csCode;
   UInt32            csParam[5];
};

typedef struct DAVCtlParam DAVCtlParam, *DAVCtlParamPtr;
```

The DAV audio driver calls accessible to third-party software are discussed in the next sections.

davSystemSubframe

The `davSystemSubframe` call is made by a client of the DAV audio driver to set the subframe to be used for the system sound input or output. The input driver and output sifter always use only one of the four available subframes. By default, they use subframe 0, which supports the Macintosh sound ASIC.

PARAMETERS

csParam[1] `DAVSubFrame`

csParam[2] `DAVIOSpec`

The `DAVSubFrame` parameter is passed to the routine for control and returned by the routine for status reporting. Its declaration and available values are given on page 316.

Input and output frame masks are the following:

```
   kInputSubFrameMask   = 0x00FF,
   kOutputSubFrameMask  = 0xFF00
};
```

The available `DAVIOSpec` values are given on page 318.

davActiveChannels

The `davActiveChannels` call is made by a client of the DAV audio driver to set the active subframes on the audio part of the DAV interface. The `activeSubFrames` parameter specifies which subframes to activate. The `DAVSubFrame` parameter is passed to the routine for control and returned by the routine for status reporting.

PARAMETER

`csParam[1] DAVSubFrame`

davPlayThrough

The `davPlayThrough` call is made by a client of the DAV audio driver to set or get the current DAV playthrough state. Playthrough allows a DAV card developer to drive audio input on one or more of the available subframes and let it be mixed with the system output on subframe 0. For example, a video teleconferencing card could drive the incoming audio from the remote caller as input on subframe 1. If playthrough is enabled for subframe 1, the DAV audio driver mixes the audio with the system sound and plays it through the built-in speaker. The only valid subframes enabled for playthrough are input subframes 1-3.

The `DAVSubFrame` parameter is passed to the routine for control and returned by the routine for status reporting.

PARAMETER

`csParam[1] DAVSubFrame`

dmaInstallIntProc

The `dmaInstallIntProc` call is made by a client of the DAV audio driver to register a handler for hardware interrupts.

```
typedef pascal void (*DAVAudioProcPtr) (short         *buff,
                                        UInt32        numBytes,
                                        DAVSubFrame   format,
                                        long          refCon);

enum {
   uppDAVAudioProcInfo = kPascalStackBased
    | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short *)))
    | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(UInt32)))
    | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(DAVSubFrame)))
    | STACK_ROUTINE_PARAMETER(4, SIZE_CODE(sizeof(long)))
};
```

The interrupt handler will be called when the hardware buffer needs to be filled (for output) or the hardware buffer is full (during input). There can be more than one handler installed for either input or output. For example, both the sound input driver and a DAV-compatible application may have handlers installed for input. Both would be called when the hardware buffer is full. The `io` parameter specifies whether the client wants to receive interrupts for the output channel, input channel, or both.

When the `DAVAudioProc` is called, it is passed a pointer to the data in `buff`, the number of bytes in the buffer in `numBytes`, and a format specification in `format`. The format specification is a bitmap of the active DAV subframes.

For example, if the buffer contains samples from both subframe 0 and subframe 1, `format` will be `0x3 (kDAVSubframe0 + kDAVSubframe1)`. Because the audio samples from the various subframes are interleaved in the buffer, interrupt handlers should use this information to parse the buffer as necessary.

PARAMETERS

`csParam[1]` (`DAVAudioProcPtr`) `callbackProc`

`csParam[2]` (`DAVIOSpec`) `io`

`csParam[3]` (`long`) `refCon`

dmaRemoveIntProc

The `dmaRemoveIntProc` call is made by a client to remove a previously installed interrupt handler. Make this call when your code no longer needs to receive hardware interrupts.

PARAMETERS

`csParam[1]` (`DAVAudioProcPtr`) `callbackProc`

`csParam[2]` (`DAVIOSpec`) `io`

dmaActive

The `dmaActive` control call is made by a client to start or stop the audio hardware. A status call can be used to determine if the input and output DMA channels are active. The `io` parameter specifies the output channel, input channel, or both.

PARAMETER

`csParam[1]` (`DAVDMAStatus`) status (current state)

The declaration and possible values for `DAVDMAStatus` are given on page 318.

IMPORTANT

Because the system software keeps track of multiple instantiations, each call to activate the audio hardware must be matched with a later call to deactivate it. ▲

cdcAuxBits

The `cdcAuxBits` control call is made by a client to transmit the 20 auxiliary bits to the Macintosh audio codec. The status call will return the last 20 bits sent to the codec. These status bits are used by the driver to control the codec on subframe 0. They can be used by DAV card developers to configure or receive device information from the card.

The `csParam[1]` parameter is used for control; status bits are returned in `csParam[2]`.

PARAMETERS

`csParam[1]` (`UInt32`) control bits

`csParam[2]` (`DAVSubFrame`) status subframe

Video Hardware Interface

Macintosh computers whose model designations end in AV support video I/O, including compatibility with NTSC, PAL, and SECAM formats. This capability lets expansion cards capture and generate a wide range of video data. It also lets software support advanced video processing features and multimedia capabilities. The DAV interface in the second generation of Power Macintosh computers provides access to the video data stream for PCI expansion cards.

This chapter helps you understand the video portion of the DAV interface by providing a general description of the video signal-processing features and circuitry in the second generation of Power Macintosh computers. For more detailed technical information, see the Apple Developer Notes that are issued with each product release.

Video Circuits

Models in the second generation of Power Macintosh computers have a built-in video subsystem that incorporates the features of the AV cards used with the first-generation and adds a few enhancements. The video subsystem handles video input and output, mixes video with computer graphics, and supports a wide variety of video monitors, including television monitors using either NTSC or PAL format.

The video subsystem is made up of an interface to the processor bus, 2 MB or 4 MB of VRAM, a video stream to the monitor, a second video stream to the video output, and an input video stream. Unlike the NuBus-based Power Macintosh computers, the PCI-based models carry both input and output video circuitry on the main circuit board.

Video Subsystem Chips

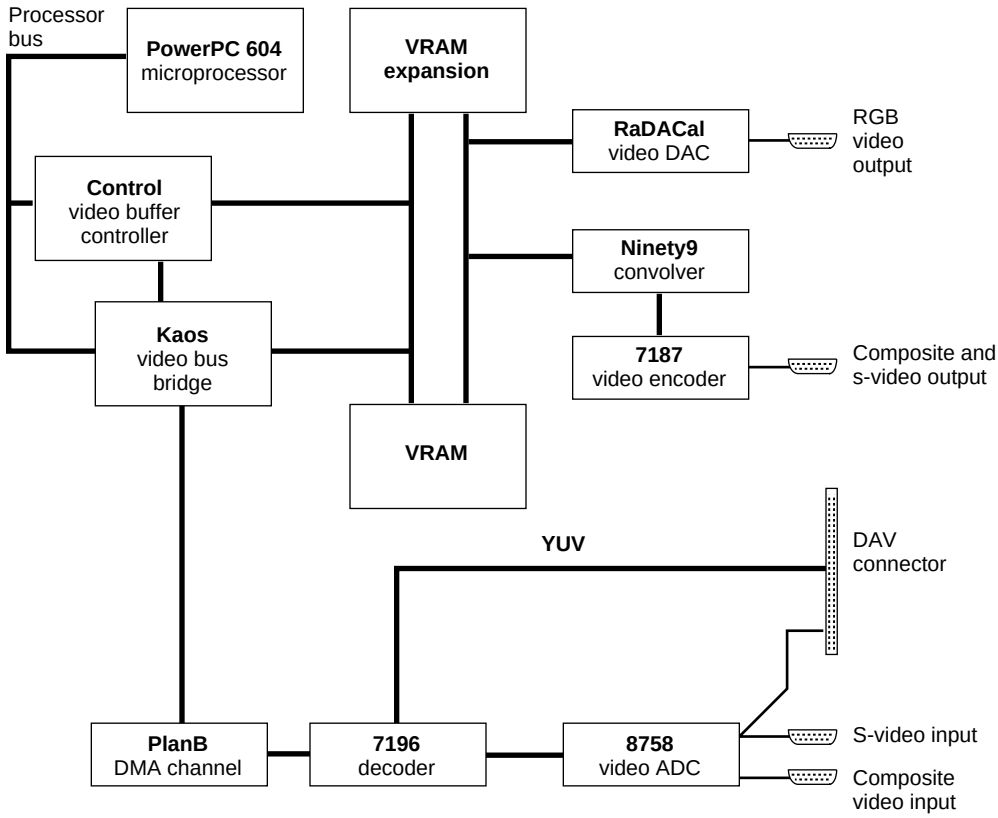
The video subsystem is implemented by the following integrated circuit chips:

- Kaos, a custom chip that provides data bus buffering between the video subsystem and VRAM and the processor bus
- Control, a custom chip that provides addressing and control for the video subsystem
- RaDACal, a high-performance DAC used for the video stream to the monitor
- Ninety9, an RGB-to-YUV converter and convolver for a second video output stream (on certain Power Macintosh models only)
- a 7187 digital video encoder for a second video output stream (on certain Power Macintosh models only)
- an 8758 analog-to-digital converter for the input video stream
- a 7196 digital video decoder, scaler, and clock generator for the input video stream
- PlanB, a DMA controller for video input data from the 7196 chip

The RaDACal chip generates the video timing for the monitor and supports the hardware cursor.

The video subsystem in a typical AV-equipped second generation Power Macintosh computer is shown in Figure 17-1.

Figure 17-1 Typical video subsystem



Video Frame Buffer

The video frame buffer is implemented by two VRAM slots, each of which accepts a 2 MB VRAM SIMM. With one SIMM installed (2 MB VRAM), the video display supports up to 24 bits per pixel on monitors up to 17 inches in size and 16 bits per pixel on a 21-inch monitor. With both SIMMs installed (4 MB VRAM), the video display supports up to 24 bits per pixel on all monitors 21 inches and smaller.

The data path to VRAM is 64 bits wide. The output data path from the VRAMs to the RaDACal high-performance DAC is 128 bits wide. The RaDACal chip provides the analog red, green, and blue signals to the monitor. The RaDACal also generates the video timing for the monitor and supports the hardware cursor.

The Ninety9 custom chip and the 7187 digital encoder produce a second video output stream, which can either mirror the graphics display or show a separate image.

With the full 4 MB of VRAM installed, the frame buffer can support separate simultaneous displays on the monitor and the second video output stream. With a

24 bits per pixel second video output stream, the buffer can simultaneously support up to 16 bits per pixel on a 21-inch monitor. With a 16 bits per pixel second video output stream, the buffer can support up to 24 bits per pixel on a 21-inch monitor.

With only 2 MB of VRAM, the computer can support only one display at a time; when the second video stream is active, the main display shuts off. With the full 4 MB of VRAM, the graphics monitor remains active at all times and the user can change the display mode without restarting the computer.

Video Input

The video input feature is implemented by the 8758 video ADC chip and the 7196 video decoder, scaler, and clock generator chip. The video data can be stored either in the video display buffer (VRAM) or in an off-screen pixel map in main memory. Video data transfers are DMA transfers and are controlled by a DBDMA engine in the PlanB chip.

When video input data is sent to the display buffer, it can be clipped with a 1-bit-per-pixel clip mask using the DBDMA read channel. This mode of data transfer provides a video playthrough mode with clipping of obscured regions and menus as well as simple titles.

When video input data is stored in a pixel map in main memory, software can perform any required clipping and blending by using the CopyBits routine in QuickDraw when moving the pixel map to a visible region of the display. The video input channel preserves the alpha channel so that software can perform alpha blending.

The PlanB chip provides two DBDMA channels for the 7196 decoder chip: a DBDMA write channel and a DBDMA read channel. The DBDMA write channel takes data from the pixel FIFO in the 7196 chip, attaches an appropriate DMA address, and performs a PCI write operation. The DBDMA read channel reads the 1-bit-per-pixel clipping mask from main memory.

The 7196 digital video decoder can also accept digital video input data from the DAV connector.

Second Stream Video Output

In addition to the main graphics display, certain Power Macintosh models supply a second video stream for an NTSC or PAL television monitor. The second video stream can either duplicate the main display (mirror mode) or display an independent image. With the full 4 MB of VRAM, the graphics monitor remains active while the second video stream is active.

The second video stream is a high-quality interlaced video signal with convolution to reduce interlace flicker. It supports television monitors using either NTSC or PAL format.

The second video stream is generated by the Ninety9 convolver chip and the 7187 video encoder chip. The Ninety9 chip converts the video data from RGB color space to YUV format and then performs convolution on the data in YUV color space. The YUV digital video format, also known as YCrCb, is described in CCIR *Recommended Standard 601-2,* listed in “Other Publications” beginning on page xxv. The 7187 encoder takes square

pixels in YUV format from the Ninety9 chip and encodes them into composite video in either NTSC or PAL format.

User Interface to Video I/O

Power Macintosh models with AV capabilities provide independent monitor and video I/O connectors. The user can connect a monitor, an input device such as a videocam, and an output device such as a VCR simultaneously. The monitor connector is a DB-15 type, compatible with most Apple monitors. The video input and output connectors accept adapter cables, supplied with the computers, that terminate in RCA connectors. These connectors are compatible with standard television sets, videocams, videodisc players, and VCRs. In the case of Power Macintosh models, the input cable connects the signal pin of an RCA socket to the luminance pin of the video input connector; the output cable connects the composite video signal pin of the AV card's video output connector to the signal pin of an RCA plug.

Video Monitor Support

The AV video circuits in PCI-based Power Macintosh computers can support mixed video and graphics in full 24-bit color (using a 32-bit pixel storage format) on small and medium-sized monitors and in 16-bit or 8-bit color on larger monitors. Table 17-1 list the ***color depths*** (the number of bits in which the color or grayscale value of each pixel can be stored) available with Apple monitors with 2 MB and 4 MB of VRAM.

Table 17-1 Color depths supported on Apple monitors

Apple monitor family	Resolution, horiz. x vert.	Refresh rate, Hz	Color depths supported	
			2 MB VRAM	4 MB VRAM
12-inch RGB	512 x 384	60	8, 16, 32	8, 16, 32
13- or 14-inch RGB Hi-Res and 12-inchMonochrome	640 x 480	67	8, 16, 32	8, 16, 32
VGA	640 x 480	60	8, 16, 32	8, 16, 32
	800 x 600	75	8, 16, 32	8, 16, 32
	1024 x 768	75	8, 16	8, 16, 32
Full-page Monochrome	640 x 870	75	8	8
Full-page RGB	640 x 870	75	8, 16,	8, 16, 32
16-inch Color	832 x 624	75	8, 16, 32	8, 16, 32
19-inch Color	1024 x 768	75	8, 16	8, 16, 32

Table 17-1 Color depths supported on Apple monitors (continued)

Apple monitor family	Resolution, horiz. x vert.	Refresh rate, Hz	Color depths supported 2 MB VRAM	4 MB VRAM
Two-page Monochrome	1152 x 870	75	8	8
Two-page RGB	1152 x 870	75	8, 16	8, 16, 32
Multiple Scan 15	640 x 480	67	8, 16, 32	8, 16, 32
	832 x 624	67	8, 16, 32	8, 16, 32
Multiple Scan 17	640 x 480	67	8, 16, 32	8, 16, 32
	832 x 624	75	8, 16, 32	8, 16, 32
	1024 x 768	75	8, 16	8, 16, 32
Multiple Scan 20	640 x 480	67	8, 16, 32	8, 16, 32
	832 x 624	75	8, 16, 32	8, 16, 32
	1024 x 768	75	8, 16	8, 16, 32
	1152 x 870	75	8, 16	8, 16, 32
	1280 x 1024	75	8	8, 16
NTSC Safetitle	512 x 384	60	8, 16, 32	8, 16, 32
NTSC Fullframe	640 x 480	60	8, 16, 32	8, 16, 32
PAL Safetitle	640 x 480	50	8, 16, 32	8, 16, 32
PAL Fullframe	768 x 576	50	8, 16, 32	8, 16, 32

Video Data Characteristics

This section describes the characteristics of video data in Power Macintosh computers, including the modes in which video data is transferred across the DAV interface and the ways it is organized as a result.

Transfer Modes

Video data transfer across the DAV interface can take place in any one of four modes depending on whether the Macintosh system or the expansion card controls the clock, synchronization, and data signals. These modes are the following:

- Mode 0: the computer controls all signals. Data flows from the video input source to system memory regardless of whether or not an expansion card is present. The card can capture data but does not drive data. This is the default mode.
- Mode 1: the expansion card uses clock and synchronization signals from the system to drive data into the system.

- Mode 2: like mode 1, except that the expansion card supplies clock and synchronization signals.
- Mode 3: the expansion card uses clock signals from the system to generate synchronization signals and drive data into the system.

The signal sources and line levels that characterize these modes are summarized in Table 17-2, where *System* indicates that the source is the Macintosh system and *Card* indicates that the source is a PCI expansion card. For information about the signals in the DAV interface, see Chapter 14, "Accessing AV Data."

Table 17-2 Data transfer mode characteristics

Signals	Mode 0	Mode 1	Mode 2	Mode 3
Clocks: LLClk and CREFB	System	System	Card	System
Horizontal and vertical synchronization	System	System	Card*	Card*
Data: Y[7–0] and UV[7–0]	System	Card	Card	Card
Control line level: DIR	Low†	High	High	High

* Interlaced synchronization is not required.
† Line pulled low by 1 kΩ resistor on the logic board.

Mode 1, in which the expansion card uses clock and synchronization signals from the system, may operate in either of two ways:

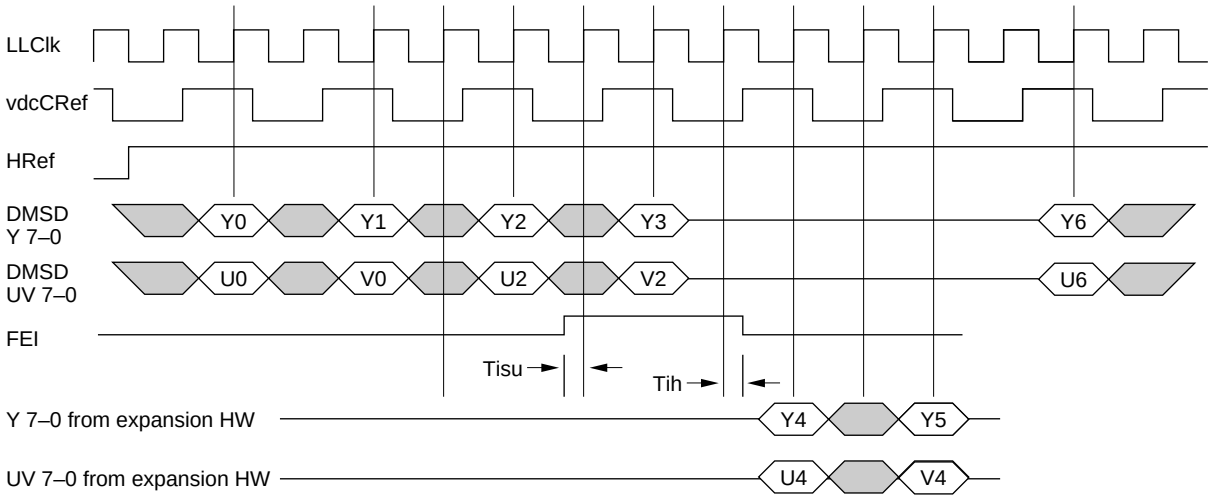
- Mode 1a: the system synchronizes its timing signals to a current video input signal.
- Mode 1b: the system generates stable timing signals without an external reference.

For example, a videoconferencing expansion card would use mode 1a to process a video signal from a remote source, whereas a video decompression card might use mode 1b to generate a video signal from stored data using timing information provided by the system.

Mode Switching

The data transfer process can be switched between mode 0 (the default mode) and mode 1 on a field-by-field or pixel-by-pixel basis, by switching the direction of data flow with the DIR line. Figure 17-2 shows the required timing relations.

Figure 17-2 Timing for switching between data transfer modes 0 and 1



The timing parameters Tisu and Tih shown in Figure 17-2 are described in Table 17-5 on page 334.

When an accessory card invokes mode 1, the system automatically operates in mode 1a. To switch to mode 1b, an accessory card must reprogram the video decoder chip as shown in Table 17-3.

Table 17-3 Decoder programming for data transfer mode 1b

Register	Bits	Value and description
$0D	D7	VTRC = 0; TV mode
$0E	D7	HPLL = 1; PLL circuit open and horizontal frequency fixed
$0F	D7	AUFD = 0; field selection by FSEL bit
$0F	D6	FSEL = 0 for 50 Hz timing; FSEL = 1 for 60 Hz timing
$10	D1, D0	VNOI1,0 = 10; free-running mode

If an external video input signal is present during mode 1b, the decoder ignores it. If it is necessary for an expansion card to switch back to mode 1a while a video signal is present, the card must wait for at least two video fields after switching so that the decoder can synchronize its timing to the signal.

Switching between mode 0 and either mode 2 or mode 3 is done by programming bits in the 7196 decoder chip. You can do this by using the video programming function described in “VDSetVideoHWState” on page 340. The bit values are shown in Table 17-4, where *x* indicates a bit value that is ignored.

Table 17-4 Mode control bit values

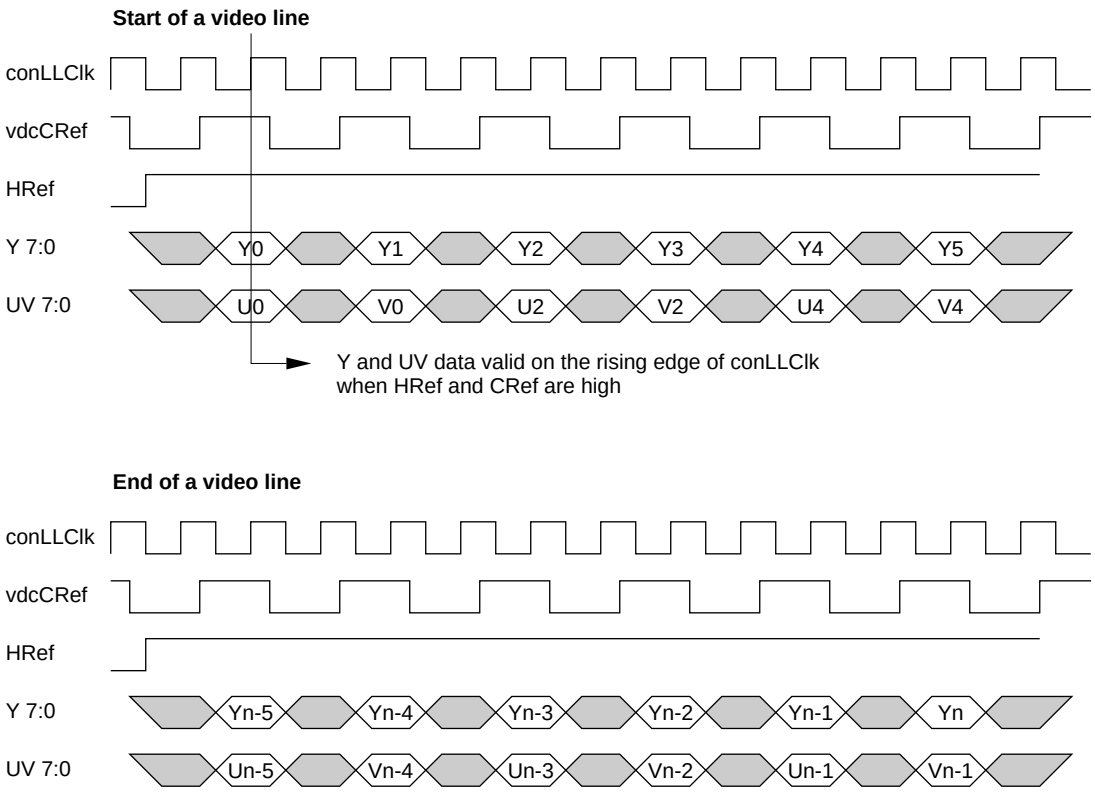
Control bits	Mode 0	Mode 1	Mode 2	Mode 3
OEYC	1	*x*	*x*	*x*
OEHV	1	1	0	0
OECL	1	1	0	1

Besides setting these bit values, the expansion card must set the direction of data flow by controlling the DIR line, as shown in Table 17-2, earlier in this chapter.

Data Organization

Digital video data is organized internally into lines and fields. A video line occurs while HRef is high, and a blanking interval occurs while HRef is low, as shown in Figure 17-3.

Figure 17-3 Video line timing



A video field is defined by the falling edge of the VS signal. For further information about field timing, see the Philips *SAA7196 Digital Video Decoder, Scaler, and Clock Generator* data sheet, listed in "Other Publications" beginning on page xxv.

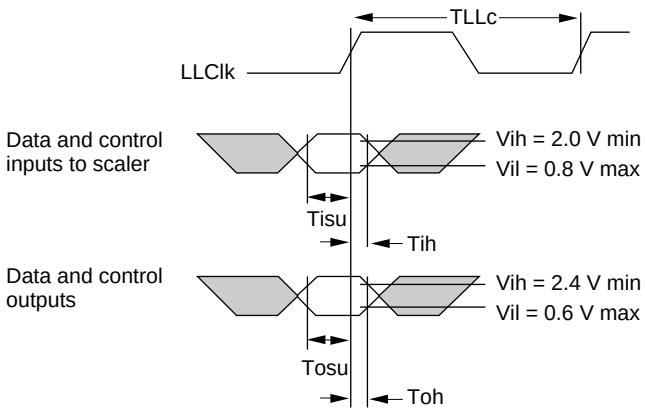
The system clock, shown as LLClk in Figure 17-3, runs at twice the video sample rate. When the Macintosh system controls the video stream (transfer mode 0), this rate is 640 samples per line for 60 Hz video and 768 samples per line for 50 Hz video. When an expansion card controls the video stream (transfer mode 1, 2, or 3), the number of samples per line should be an even number greater than 2.

Regardless of whether video data comes from the Macintosh system or from an expansion card, each pixel must be defined by both Href and vdcCref being high at the rising edge of LLClk. The exact timing relations are described in the next section.

Control Timing

When an expansion card generates control signals for the video part of the DAV interface, they must conform to the timing relations shown in Figure 17-4.

Figure 17-4 Control signal timing



The parameter values shown in Figure 17-4 must conform to the timing limits shown in Table 17-5. Besides observing the limits shown in Table 17-5, an expansion card may not load the video control and data lines by more than 20 pF.

Table 17-5 Control timing limits

Symbol	Parameter	Minimum (ns)	Maximum (ns)
TLLc	CREFB cycle time	31	45
Tisu	Input setup	11	n. a.
Tih	Input hold	3	n. a.
Tosu	Output setup	10	n. a.
Toh	Output hold	4	n. a.

Video Programming

This chapter describes the software that Apple supplies with the second generation of Power Macintosh computers to control video input data appearing at the DAV interface. For information about video signals and monitor compatibility, see Chapter 17, "Video Hardware Interface."

Video Digitizer Control

PCI expansion cards that connect to the DAV interface need some of the same services that a video digitizer offers to its clients: control of the digitizing and scaling hardware plus the ability to control the playback of information. To support these functions, Apple provides a set of externally-available extensions to the internal Macintosh video input digitizer programming interface.

These extensions, described in this chapter, let PCI cards control many parts of the built-in Power Macintosh video digitizing hardware. They comprise a group of calls to Apple's video digitizer control software that set DAV-specific features, such as the DAV mode, and return information about the hardware, such as the hardware implementation and the register values in certain chips.

In addition, the video digitizer implements suspend and resume features that permit multiple connections to it. Support for multiple video instantiations lets software gain access to the video digitizing and playback hardware while coexisting with other software that might be using the same hardware. Alternatively, the video digitizer provides external clients with a way of requesting exclusive use of the video digitizing hardware.

Video Software Architecture

To gain access to the video input features of the built-in hardware, a PCI expansion card must open an instance of the Macintosh video digitizer. Opening a digitizer instantiation gives the DAV interface access to the video digitizing and playback hardware through the programming interface of the QuickTime video digitizer component. This component has type 'vdig'. For general information about Macintosh components see *Inside Macintosh: More Macintosh Toolbox*, listed in "Apple Publications" beginning on page xxiii.

Control Features

DAV software can modify the behavior of the on-board video digitizer by using the Component Manager's targeting method. The new video digitizer instance will receive all calls destined for the on-board video digitizer; it can then modify the calls, use the on-board digitizer, or work both ways. In addition, it can request exclusive access to the hardware through the `VDRequestDAVExclusiveAccess` call.

IMPORTANT
Software developers should not abuse the feature that provides exclusive access to the video input hardware. The Macintosh default video digitizer instance will not issue exclusive access requests; this access type is provided only to third-party developers, so that they can guarantee their access to the hardware. However, they need to realize that this might be to the detriment of other clients of the hardware. Exclusive access is discussed more fully in the next section. ▲

Multiple Digitizer Instantiations

The video digitizer component can be put in a suspended state, where it responds to client requests but cannot touch the hardware. Once out of that state the component behaves in a normal way. In this manner, multiple instantiations of the digitizer component can coexist with each other. When one needs access to the hardware, all the others are put in the suspended state.

A DAV interface component handles the arbitration between multiple instantiations of the video digitizer. Third-party software does not need to open an instance of the DAV component directly; instead, when it opens the video digitizer Mac OS opens the DAV component. The video digitizer registers with the DAV component by opening an instance of that component. Thereafter, the digitizer can request or release access to the hardware.

Exclusive Access

A video digitizer requests access (using `DAVRequestAccess`) when it needs control of the hardware. This call succeeds if no other software has locked up access to the hardware. If some other digitizer instance has locked access the call will fail. A digitizer instance can insure that it gains access to the hardware by specifying in its parameters that it wants exclusive access. This forces all other digitizer instances into the suspended state, regardless of their type of access to the hardware. Having exclusive access does not mean that the digitizer instance will keep ownership of the hardware until it releases it. Another digitizer instance might later request exclusive access and automatically gain it. In the current version of the DAV component, there is no arbitration among multiple exclusive access requests.

When the DAV component receives a request call, it suspends all other video digitizer instances unless they have a lock on the hardware. It does this by making a `VDSetCurrentState` video digitizer call with a `suspend` parameter.

When a video digitizer does not need exclusive or locked access to the hardware, it sends a `DAVReleaseLock` call to the DAV component. This call sets the digitizer's access type to the shared state. In that state, any other digitizer instance can request and gain access to the hardware.

Video Digitizer Programming Interface

This section describes the routines in the Macintosh video digitizer control software that are available to third-party programs.

VDSetDAVSignalSource

The `VDSetDAVSignalSource` routine instructs the built-in video digitizer component to set the source of various video signals at the the DAV interface. For technical details of the available DAV video signals and their sources, see Chapter 17, "Video Hardware Interface."

```
pascal VDComponentError VDSetDAVSignalSource
                                (VDComponent ci,
                                 DAVSignalSource signalSource);
```

The `DAVSignalSource` data type identifies the source of signals at the DAV interface; they are either provided by the Macintosh hardware or externally by another device connected to the DAV interface. The relevant signals are

- clock signals
- synchronization signals
- YUV data

```
typedef UInt32 DAVSignalSource;
enum {
   kDAVSignalSourceOnBoardAllSignals = 0, // No external signals
   kDAVSignalSourceOnBoardClocksAndSyncs = 1, // YUV external
   kDAVSignalSourceOnBoardNoSignals = 2, // All signals external
   kDAVSignalSourceOnBoardClocks = 3, // Syncs & YUV external
};
```

VDGetDAVSignalSource

The `VDGetDAVSignalSource` routine gets the current source of signals at the DAV interface. It uses the `DAVSignalSource` data type described in the previous section.

```
pascal VDComponentError VDSetDAVSignalSource
                                (DComponent ci,
                                 DAVSignalSource *signalSource);
```

VDSetActivityState

The `VDSetActivityState` routine sets the activity state of an instantiation of the Macintosh video digitizer to suspended or active. For a discussion of suspended and active instantiations, see "Multiple Digitizer Instantiations" on page 337.

```
pascal VDComponentError VDSetActivityState
                                (VDComponent ci,
                                 VDIGActivityState activityState);

typedef UInt32 VDIGActivityState;
enum {
   kVDIGActivityStateSuspended = 0, // 'vdig' is suspended
   kVDIGActivityStateActive = 1,    // 'vdig' is active
};
```

VDGetActivityState

The `VDGetActivityState` routine returns the activity state of the video digitizer. It uses the `VDIGActivityState` data type described in the previous section.

```
pascal VDComponentErrorVDGetActivityState
                                (VDComponent ci,
                                 VDIGActivityState *activityState);
```

VDGetHWImplementation

The `VDGetHWImplementation` routine returns information about the hardware implementation of the video input subsystem.

```
pascal VDComponentError VDGetHWImplementation
                                (VDComponent ci,
                                 VDIGHWImplementation *hwInfo);

typedef UInt32 VDIGHWImplementation;
enum {
   kVDIGHW7186And7191B = 0,   // Quadra 840AV and 660AV
   kVDIGHW7194 = 1,           // 1st generation Power Macintosh
   kVDIGHW7196AndPlanB = 2,   // 2nd generation Power Macintosh
   kVDIGHW???? = 3,           // Reserved
};
```

VDGetVideoHWState

The `VDGetVideoHWState` routine retrieves the actual programmed values of the video chip registers. The information returned can be analyzed only in the context of the actual hardware implementation.

```
pascal VDComponentError VDGetVideoHWState
                                        (VDComponent ci,
                                         VDIGHWSelector hwSelector,
                                         ByteCount *size,
                                         void *storage);
```

The `hwSelector` parameter identifies a particular video chip, which can be an IIC (inter-IC control) chip or another type. Mac OS versions for future Macintosh hardware implementations may respond to additional `VDIGHWSelector` values. The currently defined values are

```
typedef UInt32 VDIGHWSelector;
enum {
   kVDIGHWSelector7186VDCC = 0,
   kVDIGHWSelectorCivicInfo,
   kVDIGHWSelector7191BDMSD,
   kVDIGHWSelector7194DESC,
   kVDIGHWSelector7196DESCPro,
   kVDIGHWSelectorPlanB,
};
```

If the `storage` parameter is nil, `VDComponentError` returns in `size` the number of bytes required to store the data. A client can then request the information by passing a valid pointer. In the case of an IIC device, the format of the data returned is as follows:

```
struct IICInfo {
   UInt8    iicAddress;       // Address or subaddress of IIC chip
   UInt32   iicRegisterCount; // Number of IIC registers
   UInt8    iicValues[1];     // A variable-length array of
                              // iicRegisterCount size
}
```

VDSetVideoHWState

The `VDSetVideoHWState` routine updates the video digitizer to new values. It is not recommended for a client to write register values directly to the hardware, bypassing the video digitizer. If the client does write to the hardware, however, it can use `VDSetVideoHWState` to update the video digitizer to the new values. The `VDSetVideoHWState` routine uses the `VDIGHWSelector` data structure described in the previous section.

```
pascal VDComponentError VDSetVideoHWState
                                    (VDComponent ci,
                                     VDIGHWSelector hwSelector,
                                     const void *storage);
```

VDGetDAVInstance

The `VDGetDAVInstance` routine returns the DAV component instance.

```
pascal VDComponentError VDGetDAVInstance
                                    (VDComponent ci,
                                     DAVComponent *davComponent);
```

VDRequestDAVExclusiveAccess

The `VDRequestDAVExclusiveAccess` routine instructs the video digitizer to always request exclusive access when using the `DAVRequestAccess` routine. The `DAVRequestAccess` routine is described on page 341.

```
pascal VDComponentError VDRequestDAVExclusiveAccess
                                    (VDComponent ci,
                                     Boolean exclusiveAccess);
```

If the `exclusiveAccess` parameter is `true`, the digitizer will always request exclusive access. If it is false (the default value), the digitizer will always request shared access.

DAV Component Programming Interface

The DAV component in the Mac OS video digitizer control software provides arbitration resources to allow multiple video digitizer instances to work with each other, as explained in "Multiple Digitizer Instantiations" on page 337. This section describes the routines in the DAV component that are available to third-party programs.

DAVRequestAccess

The `DAVRequestAccess` routine can be used by a DAV client to request access to the hardware. The `accessLevel` parameter indicates what kind of access is needed:

- shared: Access is granted if no other client has shared, locked or exclusive access.
- locked: Access is granted if no other client has locked or exclusive access.
- exclusive: Access is always granted.

If there are multiple clients, the DAV component will notify the other clients that they have lost access to the hardware by calling `VDSetCurrentState` with a `suspended`

parameter. If access can not be granted (because it is locked or exclusive), the routine will return an error.

If a client loses access to the hardware, it must request it again when it needs to access the hardware.

```
pascal DAVComponentError DAVRequestAccess
                                (DAVComponent ci,
                                 DAVAccessLevel accessLevel);

typedef UInt32 DAVAccessLevel;
enum {
   kDAVAccessLevelShared,
   kDAVAccessLevelLocked,
   kDAVAccessLevelExclusive
};
```

DAVReleaseLock

When a client has requested and been granted locked access, it can then restore its access level to `kDAVAccessLevelShared` by calling `DAVReleaseLock`. It will still own the access to the hardware, but in shared mode.

```
pascal DAVComponentError DAVReleaseLock (DAVComponent ci);
```

DAVGetVDIGInstance

The `DAVGetVDIGInstance` routine lets video digitizer give the DAV component its instance, so that the DAV component can call the `VDSetCurrentState` routine.

```
pascal DAVComponentError DAVGetVDIGInstance
                                (DAVComponent ci,
                                 VDComponent vdigInstance);
```

DAVSetVDIGInstance

The `DAVSetVDIGInstance` routine sets the current video digitizer instance that is associated with a DAV connection.

```
pascal DAVComponentError DAVSetVDIGInstance
                                (DAVComponent ci,
                                 VDComponent *vdigInstance);
```

Development Tools

This appendix describes the developer's kit that Apple provides for designers of PCI expansion cards and drivers compatible with the second generation of Power Macintosh computers.

The PCI Card Device Driver Kit contains documentation, tools, and sample code that can help you with these tasks:

- designing PCI expansion cards and hardware components for use with Power Macintosh computers
- writing the Open Firmware code for Macintosh-compatible PCI cards
- writing device drivers, system extensions, and application software to be used with Macintosh-compatible PCI cards

This preliminary draft of *Designing PCI Cards and Drivers for Power Macintosh Computers* is currently offered at no cost to interested hardware and software developers. A final kit, containing a full complement of C-based tools and sample code, is being prepared. For details and availability, contact AppleLink address APPLE.PCI.

Contents of the Device Driver Kit

The Device Driver Kit contains documentation, tools, and sample code. Parts of the kit are specific to the Macintosh implementation of the PCI and Open Firmware standards; Apple supplies these materials with the kit. Other parts are available from Apple or third parties.

Parts Supplied With the Kit

The PCI Card Device Driver Kit contains *Designing PCI Cards and Drivers for Power Macintosh Computers* and a Macintosh-compatible CD-ROM disk. The disk contains software development tools and text files of sample code designed to run on a Power Macintosh computer. The contents of the disk can be used with the Macintosh Programmer's Workshop (MPW) or with Metrowerks Code Warrior. The sample code files can also be read by TeachText and other Macintosh word processors.

Tools

The tools on the disk include a Forth compiler, a tool that tokenizes Forth code and builds a configuration ROM image, and miscellaneous utilities.

Code Files

The code files on the disk contain the C header files and libraries required to develop native drivers for the second generation of Power Macintosh computers. Some of the contents of these files are listed in "Header Files," below.

The developer kit code files also contain the C sources for a number of sample drivers for Macintosh PCI devices, plus other useful code examples.

Parts Not Included in the Kit

The tools and code samples that Apple supplies with the PCI Card Device Driver Kit can be used with the Macintosh Programmer's Workshop (MPW). Since most Macintosh developers already have MPW, it is not included in the kit. You can obtain MPW from APDA at the address listed on page xxv.

Similarly, the Apple books *Designing Cards and Drivers for the Macintosh Family,* third edition, and *Inside Macintosh: Devices* explain the general software requirements for drivers compatible with Macintosh computers. These books are extremely useful to any programmer writing a driver for a Macintosh-compatible PCI device.

Essential parts of the PCI Card Device Driver Kit for Power Macintosh Computers not supplied by Apple include the following documents:

- *PCI Local Bus Specification,* Revision 2.0, by the PCI Special Interest Group
- *PCI Bus Binding to IEEE 1275-1994,* available by contacting APPLE.PCI
- *1275-1994 Standard for Boot (Initialization, Configuration) Firmware* by the Institute of Electrical and Electronics Engineers (IEEE)
- *ANSI/IEEE X3.215-199x Programming Languages—Forth,* by the American National Standards Institute (ANSI)
- *Writing FCode Programs for PCI,* by FirmWorks

These documents are an integral part of the kit; it is difficult to design Macintosh-compatible PCI cards without their help. For information about obtaining them, see "Supplementary Documents" beginning on page xxiii.

Header Files

The PCI Card Device Driver Kit contains a large number of C 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 A-1 lists them and gives references to the sections of this book where each file's content is discussed.

Table A-1 Header files for Macintosh PCI development

File name	Book reference
`Devices.h`	Chapter 7, "Writing Native Drivers"
`DriverServices.h`	Chapter 9, "Driver Services Library"
`Gestalt.h`	"Driver Gestalt" beginning on page 90
`Interrupts.h`	"Interrupt Management" beginning on page 190
`Kernel.h`	Chapter 9, "Driver Services Library"
`NameRegistry.h`	Chapter 8, "Macintosh Name Registry"
`PCI.h`	Chapter 10, "Expansion Bus Manager"
`Video.h`	Chapter 11, "Graphics Drivers"

Table A-2 details the functions and data structures that the header files listed in Table A-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 A-2 PCI-related functions and data structures

Function or data structure	Header file	Page
`AbsoluteDeltaToDuration`	`DriverServices.h`	211
`AbsoluteDeltaToNanoseconds`	`DriverServices.h`	211
`AbsoluteTime`	`DriverServices.h`	210
`AbsoluteToDuration`	`DriverServices.h`	210
`AbsoluteToNanoseconds`	`DriverServices.h`	210
`AddAbsoluteToAbsolute`	`DriverServices.h`	211
`AddAtomic`	`DriverServices.h`	213
`AddAtomic8`	`DriverServices.h`	213
`AddAtomic16`	`DriverServices.h`	213
`AddDurationToAbsolute`	`DriverServices.h`	211
`AddNanosecondsToAbsolute`	`DriverServices.h`	211
`AddressRange`	`Kernel.h`	180
`BitAndAtomic`	`DriverServices.h`	213
`BitAndAtomic8`	`DriverServices.h`	213
`BitAndAtomic16`	`DriverServices.h`	213
`BitOrAtomic`	`DriverServices.h`	213

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
BitOrAtomic8	DriverServices.h	213
BitOrAtomic16	DriverServices.h	213
BitXorAtomic	DriverServices.h	213
BitXorAtomic8	DriverServices.h	213
BitXorAtomic16	DriverServices.h	213
BlockCopy	DriverServices.h	190
CallSecondaryInterruptHandler2	Kernel.h	205
CancelTimer	Kernel.h	213
CDDeviceCharacteristics		101
ChangeInterruptSetOptions	Interrupts.h	201
CheckpointIO	Kernel.h	185
CompareAndSwap	DriverServices.h	213
CreateInterruptSet	Interrupts.h	200
CreateSoftwareInterrupt	Kernel.h	203
CStrCat	DriverServices.h	215
CStrCmp	DriverServices.h	216
CStrCopy	DriverServices.h	215
CStrLen	DriverServices.h	216
CStrNCat	DriverServices.h	215
CStrNCopy	DriverServices.h	215
CStrNCmp	DriverServices.h	216
CStrToPStr	DriverServices.h	216
CurrentExecutionLevel	Kernel.h	175
CurrentTaskID	Kernel.h	203
DecrementAtomic	DriverServices.h	213
DecrementAtomic8	DriverServices.h	213
DecrementAtomic16	DriverServices.h	213
DelayFor	Kernel.h	212
DelayForHardware	Kernel.h	212
DeleteSoftwareInterrupt	Kernel.h	203
DeviceAddressPhysicalToLogical	DriverServices.h	221
DeviceInfoRecord		97

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
DeviceProbe	DriverServices.h	123
DriverConfigParam		94
DriverDescription	Devices.h	73
DriverFinalInfo	Devices.h	80
DriverGestaltBootResponse		96
DriverGestaltDAPIResponse		97
DriverGestaltDevTResponse		96
DriverGestaltGetLResponse		97
DriverGestaltGetVResponse		97
DriverGestaltIntfResponse		96
DriverGestaltIsOn	Devices.h	91
DriverGestaltOff	Devices.h	91
DriverGestaltOn	Devices.h	91
DriverGestaltParam		94
DriverGestaltSyncResponse		96
DriverGestaltWideResponse		97
DriverInitInfo	Devices.h	80
DriverOSRuntime	Devices.h	76
DriverOSService	Devices.h	77
DriverServiceInfo	Devices.h	77
DriverType	Devices.h	75
DurationToAbsolute	DriverServices.h	210
DurationToNanoseconds	DriverServices.h	210
ExpMgrConfigReadByte	PCI.h	229
ExpMgrConfigReadLong	PCI.h	230
ExpMgrConfigReadWord	PCI.h	229
ExpMgrConfigWriteByte	PCI.h	230
ExpMgrConfigWriteLong	PCI.h	232
ExpMgrConfigWriteWord	PCI.h	231
ExpMgrInterruptAcknowledgeReadByte	PCI.h	232
ExpMgrInterruptAcknowledgeReadLong	PCI.h	233
ExpMgrInterruptAcknowledgeReadWord	PCI.h	233

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
ExpMgrIOReadByte	PCI.h	224
ExpMgrIOReadLong	PCI.h	226
ExpMgrIOReadWord	PCI.h	225
ExpMgrIOWriteByte	PCI.h	226
ExpMgrIOWriteLong	PCI.h	228
ExpMgrIOWriteWord	PCI.h	227
ExpMgrSpecialCycleBroadcastLong	PCI.h	233
ExpMgrSpecialCycleWriteLong	PCI.h	234
FinalizationInfo		80
FindDriversForDevice	Devices.h	106
FlushProcessorCache	DriverServices.h	186
GetDataCacheLineSize	DriverServices.h	186
GetDriverDiskFragment	Devices.h	106
GetDriverForDevice	Devices.h	107
GetDriverInformation	Devices.h	116
GetDriverMemoryFragment	Devices.h	105
GetInterruptFunctions	Interrupts.h	202
GetInterruptSetOptions	Interrupts.h	201
GetIOCommandInfo	DriverServices.h	82
GetLogicalPageSize	DriverServices.h	186
GetPageInformation	Kernel.h	187
GetTimeBaseInfo	DriverServices.h	208
HigherDriverVersion	Devices.h	115
HighestUnitNumber	Devices.h	117
IncrementAtomic	DriverServices.h	213
IncrementAtomic8	DriverServices.h	213
IncrementAtomic16	DriverServices.h	213
InitializationInfo		80
InstallDriverForDevice	Devices.h	114
InstallDriverFromDisk	Devices.h	110
InstallDriverFromFile	Devices.h	112
InstallDriverFromFragment	Devices.h	109

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
InstallDriverFromMemory	Devices.h	113
InstallInterruptFunctions	Interrupts.h	131
InterruptDisabler	Interrupts.h	200
InterruptEnabler	Interrupts.h	199
InterruptHandler	Interrupts.h	199
InterruptSetMember	Interrupts.h	198
IOCommandIsComplete	DriverServices.h	69
IOPreparationTable	Kernel.h	179
LogicalAddressRange	Kernel.h	179
LookupDrivers	Devices.h	117
LowPowerMode		100
MemAllocatePhysicallyContiguous	DriverServices.h	189
MemDeallocatePhysicallyContiguous	DriverServices.h	189
MultipleAddressRange	Kernel.h	180
NanosecondsToAbsolute	DriverServices.h	210
NanosecondsToDuration	DriverServices.h	211
NumVersion		115
OpenInstalledDriver	Devices.h	111
PageInformation	Kernel.h	186
partInfoRec		99
PBDequeue	DriverServices.h	215
PBDequeueFirst	DriverServices.h	215
PBDequeueLast	DriverServices.h	215
PBEnqueue	DriverServices.h	215
PBEnqueueLast	DriverServices.h	215
PBQueueCreate	DriverServices.h	214
PBQueueDelete	DriverServices.h	214
PBQueueInit	DriverServices.h	214
PhysicalAddressRange	Kernel.h	179
PoolAllocateResident	DriverServices.h	188
PoolDeallocate	DriverServices.h	189
PrepareMemoryForIO	Kernel.h	184

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
PStrCat	DriverServices.h	215
PStrCmp	DriverServices.h	216
PStrCopy	DriverServices.h	215
PStrLen	DriverServices.h	216
PStrNCat	DriverServices.h	215
PStrNCmp	DriverServices.h	216
PStrNCopy	DriverServices.h	215
PStrToCStr	DriverServices.h	216
QueueSecondaryInterruptHandler	Kernel.h	205
RegEntryID	NameRegistry.h	140
RegEntryIter	NameRegistry.h	144
RegistryCStrEntryCreate	NameRegistry.h	142
RegistryCStrEntryLookup	NameRegistry.h	147
RegistryCStrEntryToName	NameRegistry.h	149
RegistryCStrEntryToPath	NameRegistry.h	148
RegistryEntryDelete	NameRegistry.h	143
RegistryEntryGetMod	NameRegistry.h	162
RegistryEntryIDCompare	NameRegistry.h	141
RegistryEntryIDCopy	NameRegistry.h	141
RegistryEntryIDDispose	NameRegistry.h	142
RegistryEntryIDInit	NameRegistry.h	140
RegistryEntryIterate	NameRegistry.h	145
RegistryEntryIterateCreate	NameRegistry.h	144
RegistryEntryIterateDispose	NameRegistry.h	146
RegistryEntryIterateSet	NameRegistry.h	144
RegistryEntryMod	NameRegistry.h	160
RegistryEntryPropertyMod	NameRegistry.h	161
RegistryEntrySearch	NameRegistry.h	146
RegistryEntrySetMod	NameRegistry.h	162
RegistryEntryToPathSize	NameRegistry.h	148
RegistryPropertyCreate	NameRegistry.h	151
RegistryPropertyDelete	NameRegistry.h	152

Table A-2 PCI-related functions and data structures (continued)

Function or data structure	Header file	Page
RegistryPropertyGet	NameRegistry.h	155
RegistryPropertyGetMod	NameRegistry.h	163
RegistryPropertyGetSize	NameRegistry.h	154
RegistryPropertyIterate	NameRegistry.h	153
RegistryPropertyIterateCreate	NameRegistry.h	153
RegistryPropertyIterateDispose	NameRegistry.h	154
RegistryPropertySet	NameRegistry.h	155
RegistryPropertySetMod	NameRegistry.h	163
RegPropertyIter	NameRegistry.h	152
RemoveDriver	Devices.h	115
RemoveInterruptFunctions		132
RenameDriver	Devices.h	126
ReplaceDriverWithFragment	Devices.h	126
SendSoftwareInterrupt	Kernel.h	203
SetInterruptTimer	Kernel.h	212
SetProcessorCacheMode	Kernel.h	187
SubAbsoluteFromAbsolute	DriverServices.h	211
SubDurationFromAbsolute	DriverServices.h	211
SubNanosecondsFromAbsolute	DriverServices.h	211
SynchronizeIO	DriverServices.h	188
SysDebug	Kernel.h	216
SysDebugStr	Kernel.h	216
TestAndClear	DriverServices.h	214
TestAndSet	DriverServices.h	214
UpTime	DriverServices.h	210
VerifyFragmentAsDriver	Devices.h	109
VSLNewInterruptService	Video.h	273
VSLDoInterruptService	Video.h	274
VSLDisposeInterruptService	Video.h	274

Development Tools

Glossary

address-invariant byte swapping A technique for changing data between ***big-endian*** and ***little-endian*** formats that preserves the addresses of data bytes.

address space The domain of addresses in memory that can be directly referenced by the processor at any given moment.

analog-to-digital converter (ADC) Circuitry that measures analog electrical levels and delivers the results as digital data.

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.

APDA Apple's worldwide direct distribution channel for Apple and third-party development tools and documentation products.

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.***

audio component An Apple audio configuration utility managed by the Component Manager.

audio waveform amplifier and converter (AWAC) A chip in Power Macintosh computers that combines a waveform amplifier with a digital encoder and decoder (codec) for analog sound data, including speech.

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.***

AWAC See ***audio waveform amplifier and converter.***

big-endian Used to describe data formatting in which each field is addressed by referring to its most significant byte. See also ***little-endian.***

boot driver A device driver that is used during the Open Firmware startup process. It must be written in FCode and is usually loaded from the expansion ROM on a PCI card.

bridge See ***PCI bridge.***

CCIR Comité Consultatif International Radio.

CD-ROM A digital recording medium in which data is recorded optically on plastic discs.

CFM See ***Code Fragment Manager.***

codec A digital encoder and decoder.

Code Fragment Manager (CFM) A part of the Power Macintosh system software 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.

composite video A video signal that includes both picture information (with chroma and luminance combined) and the timing and other signals needed to display it. It is the standard

signal form for communication between videocassette recorders, television sets, and other common video equipment. See also ***S-video.***

convolution The process of smoothing alternate lines of a video signal to be shown in succeeding frames for a line-interlaced display.

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.

configuration section A part of a driver that handles device configuration information. It doesn't necessarily reside in the operating system and can be an INIT or a piece of user interface code such as a CDEV or RDEV. The configuration section has access to the Macintosh Toolbox.

control section A part of a driver that controls driver functions and has access to the Macintosh Toolbox. It communicates with the ***main driver*** via Device Manager calls or interrupts.

cookie A parameter in programming that is used only to transfer a value from one routine to another.

DAC See ***digital-to-analog converter.***

Data Link Provider Interface (DLPI) The standard interface Apple uses for Open Transport drivers.

DAV See ***digital audio/video (DAV) interface.***

DAV audio driver A native PowerPC driver that Apple supplies for DAV audio signals.

DAV connector A separate connector in certain Macintosh computers that lets an expansion card access audio and video data directly instead of through the normal expansion card bus.

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 the Macintosh system software 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 PCI devices available to the system. Mac OS extracts information from the device tree to construct the device parts of the Macintosh ***Name Registry.***

digital audio/video (DAV) interface A connector in certain Macintosh models that lets expansion cards communicate directly with the system's audio and video signal streams.

digital-to-analog converter (DAC) Circuitry that produces analog electrical levels in response to digital data.

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 located in the Macintosh file system in the Extensions folder.

Display Manager A part of the Power Macintosh system software 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 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 Services Library (DSL) A CFM shared library that supplies all the system programming interfaces required by native device drivers.

Driver Loader Library (DLL) A CFM shared library extension to the Device Manager, which installs and removes drivers.

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 Macintosh startup firmware that provides access to I/O memory and manages the storage of certain information in nonvolatile RAM.

expansion ROM A ROM on a PCI accessory 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.*

expert The code that connects a family of devices to the 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 places information into the Name Registry about a family of devices.

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 pre-computed packet header for a given destination.

FCode A tokenized version of the Forth programming language, 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 A 256-bit format in which sound data is recorded.

frame buffer Memory that stores one or more frames of video information until they are displayed 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 level The execution context provided to a device driver's primary interrupt handler.

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 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 low-level code into the System 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.*

main driver The part of a driver that that resides in the operating system and does all the work of responding to the I/O command set for a particular device family.

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 is not stale.

mini-DIN An international standard form of cable connector for peripheral devices.

modifier Information associated with a name or property that his hardware- or implementation-specific, such as whether or not the name or property is saved to nonvolatile RAM.

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 I/O framework of the second generation of 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 and SPIs in Mac OS that support native run-time drivers.

noninterrupt level The execution environment for applications and other programs that do not service interrupts. Also called *task level.*

nonvolatile RAM (NVRAM) Memory, in either flash ROM or battery-powered RAM, that retains data between system startups.

NTSC An acronym for National Television System Committee, the television signal format common in North America, Japan, parts of South America, and other regions.

Open Firmware startup process The startup process by which PCI-compatible Macintosh computers with PowerPC processors recognize and configure peripheral devices connected to the PCI bus. It conforms to an IEEE standard.

Open Transport A device family that handles Apple network devices such as LocalTalk and Ethernet.

PAL An acronym for Phased Alternate Lines, the television signal format common in Western Europe (except France), China, Australia, parts of South America and the Middle East, most of Africa, and most of southern Asia.

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 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. The PowerPC 601 microprocessor is currently used in Macintosh PCI-based computers.

primary interrupt handler The part of an interrupt handler that responds directly to an interrupt. It usually queues a ***secondary interrupt handler*** to perform the bulk of the interrupt servicing.

property A piece of descriptive information associated with a node in the device tree or with a name 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.

registration The process of attaching an interrupt handler to the ***Interrupt Source Tree.***

RGB Abbreviation for *red-green-blue.* A data format for color displays in which the red, green, and blue values of each pixel are separately encoded.

RISC See ***reduced instruction set computing.***

ROM driver A driver located in the expansion ROM of a PCI 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 or contained in the expansion ROM on a PCI card. In the second generation of Power Macintosh, all run-time drivers are ***native drivers.***

scanning The process of matching a device with its corresponding driver.

scatter-gather list The set of physical address ranges corresponding to a logical address range.

SCSI Interface Module (SIM) The equivalent of a driver for devices compatible with SCSI Manager 4.3.

SECAM A French acronym for the television signal format used in France, Eastern Europe, the former Soviet Union, most of the Middle East, and many former French colonies.

secondary interrupt handler A part of an interrupt handler that is queued for execution after the primary part has responded to the interrupt. Secondary interrupt handlers execute serially when the system is not otherwise busy.

secondary interrupt level The execution context provided to a device driver's secondary interrupt handler.

SIM See ***SCSI Interface Module.***

SPI See ***System Programming Interface.***

startup firmware Code in the Macintosh ROM that implements the Open Firmware startup process.

subframe The part of an audio ***frame*** that carries one stereo channel.

S-video A video format in which chroma and luminance are transmitted separately. It provides higher image quality than ***composite video.***

System Programming Interface (SPI) A set of services in the Macintosh system software that supports hardware-related code such as drivers. See ***Application Programming Interface.***

task level see ***noninterrupt level.***

time base The model-dependent rate on which real-time timing operations are based in Power Macintosh computers.

vertical blanking task A task that the Macintosh system 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 data format for each pixel of a color display in which color is encoded by values calculated from the pixel's native red, green, and blue components. YUV format is defined by CCIR *Recommended Standard 601-2.*

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Final pages were created on an Apple LaserWriter Pro 630 printer. Line art was created using Adobe Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
George Towner

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATORS
Sandee Karr

PRODUCTION
Rex Wolf

Special thanks to Jeff Boone, Jano Banks, Clinton Bauder, Mark Baumwell, Öner Biçakçi, Bill Bruffey, Derrick Carty, Bruce Eckstein, Kit Fitzpatrick, Ian Hendry, Ron Hochsprung, Fred Janon, Holly Knight, Al Kossow, Tom Mason, Martin Minow, John Mitchell, Matthew Nelson, Jon Norenberg, Noah Price, Mike Puckett, Steve Polzin, Mark Pontarelli, Mike Quinn, Tom Saulpaugh, Erik Staats, Carl Sutton, Larry Thompson, Fernando Urbina, Allen Watson, and Tony Wingo.