

Developer Note

Macintosh PowerBook 190 Computer

Macintosh PowerBook 190
Macintosh PowerBook 190cs

Apple Computer, Inc.
© 1995 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

The Apple logo is a trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, APDA, AppleLink, AppleTalk, LaserWriter, LocalTalk, Macintosh, Macintosh Quadra, Newton, PowerBook, Power Macintosh, and ProDOS are trademarks of Apple Computer, Inc., registered in the United States and other countries.

AOCE, Apple Desktop Bus, AppleScript, Disk First Aid, Finder, Mac, Macintosh, PC Exchange, and QuickDraw are trademarks of Apple Computer, Inc.

Adobe Illustrator, Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

America Online is a service mark of Quantum Computer Services, Inc.

Classic is a registered trademark licensed to Apple Computer, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

IBM is a registered trademark of International Business Machines Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Motorola is a registered trademark of Motorola Corporation.

NuBus is a trademark of Texas Instruments.

PowerPC is a trademark of International Business Machines Corporation, used under license therefrom.

Unix is a registered trademark of Novell, Inc., in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Windows is a trademark of Microsoft Corporation, and SoftWindows is a trademark used under license by insignia from Microsoft Corporation.

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

	Figures and Tables	ix
Preface	About This Developer Note	xi
	Contents of This Note	xi
	Supplemental Reference Documents	xii
	Apple Publications	xii
	Other Publications	xiii
	Conventions and Abbreviations	xiv
	Typographical Conventions	xiv
	Standard Abbreviations	xiv
Chapter 1	Introduction	1
	Features	2
	Appearance	3
	Configurations	5
	Peripheral Devices	6
	Compatibility Issues	6
	RAM Expansion Cards	6
	Number of Colors	6
	Video Mirror Mode	7
	Sound Sample Rates	7
	Power Manager Interface	7
Chapter 2	Architecture	9
	Processor/Memory Subsystem	11
	Microprocessor	11
	RAM	11
	ROM	11
	Pratt Memory Controller IC	11
	Bus Bridge	12
	Input/Output Subsystem	12
	Whitney Peripheral Support IC	12
	Combo IC	13
	Singer IC	13
	Power Manager IC	13
	Display Controller IC	14
	Baboon Custom IC	14
	TREX Custom IC	15

Video Card	15
Keystone Video Controller IC	15
Ariel Video Output IC	15

Chapter 3 **I/O Features** 17

Internal Hard Disk Drive	18
Hard Disk Specifications	18
Hard Disk Connectors	20
Connector Location	20
Pin Assignments	21
IDE Signal Descriptions	22
Trackpad	23
Keyboard	23
Flat Panel Displays	24
Flat Panel Display Circuitry	24
Number of Colors	25
Serial Port	25
SCSI Port	26
ADB Port	28
Infrared Module	29
Sound System	29
Sound Inputs	30
Sound Outputs	30

Chapter 4 **Expansion Modules** 31

Expansion Bay	32
Expansion Bay Design	32
Expansion Bay Connector	33
Signals on the Expansion Bay Connector	34
Signal Definitions	35
Unused IDE Signals	37
Power on the Expansion Bay	37
User Installation of an Expansion Bay Device	38
Sequence of Control Signals	38
Guidelines for Developers	38
RAM Expansion	39
Electrical Design Guidelines for the RAM Expansion Card	39
Connector Pin Assignments	39
Signal Descriptions	42
Address Multiplexing	43
Banks of DRAM	44
DRAM Device Requirements	44
Expansion Card Electrical Limits	45

Mechanical Design of the RAM Expansion Card	46
RAM Card Dimensions	46
RAM Card Connector	47
Video Card	48
The Apple Video Card	48
Monitors Supported	48
Video Mirroring	49
External Video Connector	50
Monitor Sense Codes	51
Video Card Design Guide	52
Video Card Connector	52
Signals on the Video Card Connector	52
Video Card Mechanical Design	54
PCMCIA Slot	57
PCMCIA Features	57
Summary Specifications	58
Access Windows	58
Data Access	58
Signal Definitions	58
Power	59
Controller Interrupts	59

Chapter 5 **Software Features** 61

ROM Software	62
Machine Identification	62
Memory Controller Software	63
Power Manager Software	63
Display Controller Software	63
Sound Features	63
ATA Storage Devices	64
IDE Disk Mode	64
Ethernet Driver	64
Support for Function Keys	64
Smart Battery Support	64
Trackpad Support	65
System Software	65
Control Strip	66
Support for ATA Devices	66
Large Partition Support	66
64-Bit Volume Addresses	66
System-Level Software	66
Application-Level Software	67
Limitations	67
Drive Setup	67
Improved File Sharing	68

Math Library	68
QuickDraw Acceleration API	68
Display Manager	68

Chapter 6 **Large Volume Support** 69

Overview of the Large Volume File System	70
API Changes	70
Allocation Block Size	70
File Size Limits	71
Compatibility Requirements	71
The API Modifications	71
Data Structures	71
Extended Volume Parameter Block	71
Extended I/O Parameter Block	73
New Extended Function	75

Chapter 7 **Software for ATA Devices** 79

Introduction to the ATA Software	80
ATA Disk Driver	81
Drives on PC Cards	82
Drives in the Expansion Bay	83
ATA Manager	83
ATA Disk Driver Reference	83
Standard Device Routines	84
The Control Routine	84
The Status Routine	85
Control Functions	86
Status Functions	93
ATA Manager Reference	98
The ATA Parameter Block	99
Functions	104
Using the ATA Manager With Drivers	132
Notification of Device Events	132
Device Driver Loading	133
New API Entry Point for Device Drivers	134
Loading a Driver From the Media	135
Notify-all Driver Notification	135
ROM Driver Notification	136
Device Driver Purging	136
Setting the I/O Speed	138
Error Code Summary	139

Chapter 8 **PC Card Services** 143

Client Information	144
Configuration	148
Masks	155
Tuples	159
Card and Socket Status	163
Access Window Management	164
Client Registration	168
Miscellaneous Functions	170
PC Card Manager Constants	177

Glossary 181

Index 183

Figures and Tables

Chapter 1	Introduction	1
	Figure 1-1	Front view of the computer 4
	Figure 1-2	Back view of the computer 5
	Table 1-1	Models and configurations 5
Chapter 2	Architecture	9
	Figure 2-1	Block diagram 10
Chapter 3	I/O Features	17
	Figure 3-1	Maximum dimensions of the internal hard disk 19
	Figure 3-2	Connector for the internal IDE hard disk 20
	Figure 3-3	Position of the hard disk connector 20
	Figure 3-4	Keyboard, U.S. layout 23
	Figure 3-5	Keyboard, ISO layout 24
	Figure 3-6	Serial port connector 25
	Figure 3-7	ADB connector 28
	Table 3-1	Pin assignments on the hard disk connector 21
	Table 3-2	Signals on the IDE hard disk connector 22
	Table 3-3	Characteristics of the displays 24
	Table 3-4	Serial port signals 26
	Table 3-5	Signals on the SCSI connector 26
	Table 3-6	ADB connector pin assignments 28
Chapter 4	Expansion Modules	31
	Figure 4-1	Expansion bay module 32
	Figure 4-2	Expansion bay dimensions 33
	Figure 4-3	RAM expansion card 46
	Figure 4-4	Dimensions of the RAM expansion card 46
	Figure 4-5	Restricted areas on the component side of the card 47
	Figure 4-6	Video card 48
	Figure 4-7	Video connectors 51
	Figure 4-8	Dimensions of the video card 54
	Figure 4-9	Video card and 80-pin connector 55
	Figure 4-10	Video card bottom view with component restrictions 55
	Figure 4-11	Video card top view with component restrictions 56
	Figure 4-12	Video card top view 56
	Figure 4-13	Detail of EMI shield mounting holes 57

Table 4-1	Signal assignments on the expansion bay connector	34
Table 4-2	Control signals on the expansion bay connector	36
Table 4-3	Floppy disk signals on the expansion bay connector	36
Table 4-4	IDE signals on the expansion bay connector	36
Table 4-5	Unused IDE signals	37
Table 4-6	Power for the expansion bay	37
Table 4-7	Configurations of RAM banks	39
Table 4-8	Signal assignments on the RAM expansion connector	40
Table 4-9	Descriptions of signals on the RAM expansion connector	42
Table 4-10	Address multiplexing for some typical DRAM devices	44
Table 4-11	Video monitors and modes	49
Table 4-12	Signals on the video connector	50
Table 4-13	Monitor sense codes	51
Table 4-14	Signals on the video card connector	52
Table 4-15	Descriptions of the signals on the video card connector	53

Chapter 7

Software for ATA Devices 79

Figure 7-1	ATA software model	80
Table 7-1	Control functions	84
Table 7-2	Status functions	85
Table 7-3	Control bits in the <code>ataFlags</code> field	101
Table 7-4	ATA Manager functions	104
Table 7-5	Event masks	112
Table 7-6	Bits in <code>pcValid</code> field	119
Table 7-7	ATA register selectors	127
Table 7-8	Register mask bits	128
Table 7-9	Event codes send by the ATA Manager	132
Table 7-10	Input parameter bits for the old API	134
Table 7-11	Input parameter bits for the new API	134
Table 7-12	Purge permissions and responses	137
Table 7-13	ATA driver error codes	139

About This Developer Note

This developer note describes the Macintosh PowerBook 190 computer, emphasizing the features that are new or different from those of other Macintosh PowerBook computers.

This developer note is intended to help hardware and software developers design products that are compatible with the Macintosh products described in the note. If you are not already familiar with Macintosh computers or if you would simply like more technical information, you may wish to read the supplementary reference documents described in this preface.

This note is published in two forms: an online version included with the Apple Developer CD and a paper version published by APDA. For information about APDA, see “Supplemental Reference Documents.”

Contents of This Note

The information in this note is arranged in eight chapters.

- Chapter 1, “Introduction,” introduces the Macintosh PowerBook 190 computer and describes its new features.
- Chapter 2, “Architecture,” describes the internal logic of the Macintosh PowerBook 190 computer, including the main ICs that appear in the block diagram.
- Chapter 3, “I/O Features,” describes the input/output features, including both the internal I/O devices and the external I/O ports.
- Chapter 4, “Expansion Modules,” describes the expansion features of interest to developers: the expansion bay, the RAM expansion connector, the video card connector, and the PCMCIA slot.
- Chapter 5, “Software Features,” describes the new features of the ROM and system software, with the emphasis on software that is specific to this computer.
- Chapter 6, “Large Volume Support,” describes the modifications that enable the file system to support volumes larger than 4 GB.
- Chapter 7, “Software for ATA Devices,” describes the low-level program interface used by utility software for the IDE hard disk drive.
- Chapter 8, “PC Card Services,” describes the new system software that supports PC cards in the PCMCIA slot.

Supplemental Reference Documents

The following documents provide information that complements or extends the information in this developer note.

Apple Publications

Developers should have copies of the appropriate Apple reference books, including the relevant volumes of *Inside Macintosh*; *Guide to the Macintosh Family Hardware*, second edition; and *Designing Cards and Drivers for the Macintosh Family*, third edition. These Apple books are available in technical bookstores and through APDA.

For information about the PCMCIA slot, PC cards, and the PC Card Manager, developers should have a copy of *Developing PC Card Software for the Mac OS*. That book is currently available only in draft form, but it is scheduled for publication at about the time the Macintosh PowerBook 190 computer is introduced.

For information about the Device Manager and the Power Manager, developers should have a copy of *Inside Macintosh: Devices*. For information about designing device drivers for Power Macintosh computers, developers should have a copy of *Designing PCI Cards and Drivers for Power Macintosh Computers*.

For information about the control strip, developers should have the Reference Library volume of the Developer CD Series, which contains Macintosh Technical Note *OS 06 – Control Strip Modules*.

For information about earlier PowerBook models, developers should also have copies of the *Macintosh Classic II*, *Macintosh PowerBook Family*, and *Macintosh Quadra Family Developer Notes*; and *Macintosh Developer Notes*, numbers 1 through 5 and 9. These developer notes are available on the Developer CD Series and through APDA.

APDA is Apple Computer'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.

P R E F A C E

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

To supplement the information in this developer note, developers should have copies of the appropriate Motorola reference books for the MC68040 microprocessor. Software developers should have a copy of Motorola's *MC68040 Programmer's Reference Manual*. Hardware developers should have copies of Motorola's *MC68030 User's Manual*, *MC68040 User's Manual*, and *MC68040 Designer's Handbook*.

For information about the IDE hard disk drive, developers should have a copy of the ATA/IDE specification, ANSI proposal X3T10/0948D, Revision 2K or later (ATA-2).

For information about PC cards and the PCMCIA slot, developers should refer to the *PC Card Standard*. You can order that book from

Personal Computer Memory Card International Association

1030G East Duane Avenue

Sunnyvale, CA 94086

Phone: 408-720-0107

Fax: 408-720-9416

Conventions and Abbreviations

This developer note uses the following typographical conventions and abbreviations.

Typographical Conventions

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 of interest but is not essential for an understanding of the text. ◆

IMPORTANT

A note like this contains important information that you should read before proceeding. ▲

▲ WARNING

Warnings like this direct your attention to something that could cause injury to the user, damage to either hardware or software, or loss of data. ▲

Standard Abbreviations

Standard units of measure used in this note include

A	amperes	K	1024
cm	centimeters	KB	kilobytes
dB	decibels	kHz	kilohertz
GB	gigabytes	kΩ	kilohms
Hz	hertz	M	1,048,576
k	1000	mA	milliamperes

continued

Sidebar

Sidebars are used for information that is not part of the main discussion. A sidebar may contain

information about a related subject or technical details that are not required reading.

P R E F A C E

MB	megabytes	μ s	microseconds
MHz	megahertz	ns	nanoseconds
mm	millimeters	Ω	ohms
ms	milliseconds	pF	picofarads
mV	millivolts	V	volts
μ F	microfarads	VAC	volts AC

Other abbreviations used in this note include

$\$n$	hexadecimal value n
AC	alternating current
ADB	Apple Desktop Bus
API	application program interface
ATA	AT attachment
ATAPI	ATA packet interface
BCD	binary-coded decimal
CAS	column address strobe (a memory control signal)
CCFL	cold cathode fluorescent lamp
CD	compact disc
CD-ROM	compact-disc read-only memory
CIS	card information structure
CLUT	color lookup table
CMOS	complementary metal oxide semiconductor
CPU	central processing unit
CSC	color screen controller
DAC	digital-to-analog converter
DDM	driver descriptor map
DMA	direct memory access
DOS	disk operating system
DRAM	dynamic RAM
FIFO	first in, first out
FPU	floating-point unit
FSTN	film supertwist nematic (a type of LCD)
HBA	host bus adapter
HFS	hierarchical file system
IC	integrated circuit
IDE	integrated device electronics
I/O	input/output

P R E F A C E

IR	infrared
IWM	Integrated Woz Machine (a custom IC that controls the floppy disk interface)
LCD	liquid crystal display
LS TTL	low-power Schottky TTL (a standard type of device)
MMU	memory management unit
n.c.	no connection
PCMCIA	Personal Computer Memory Card International Association
PWM	pulse width modulation
RAM	random-access memory
RAMDAC	random-access memory, digital to analog converter
RAS	row address strobe
RGB	red-green-blue (a type of color video system)
rms	root-mean-square
ROM	read-only memory
SCC	Serial Communications Controller
SCSI	Small Computer System Interface
SNR	signal-to-noise ratio
SOJ	small outline J-lead package
SOP	small outline package
SRAM	static RAM
SVGA	super video graphics adapter
TDM	time-division multiplexing
TFT	thin-film transistor (a type of LCD)
TSOP	thin small outline package
TTL	transistor-transistor logic (a standard type of device)
VCC	positive supply voltage (voltage for collectors)
VGA	video graphics adapter
VRAM	video RAM

Introduction

Introduction

The Macintosh PowerBook 190 computer is a full-featured, all-in-one notebook computer with a 68040 microprocessor. Inside the computer's contoured case are a PCMCIA slot, an expansion bay for a floppy disk drive or an 8 cm CD-ROM drive, and space for a rechargeable battery.

Features

Here is a summary of the major features of the Macintosh PowerBook 190 computer. Each feature is described more fully later in this developer note.

- **Micoprocessor:** The Macintosh PowerBook 190 computer has an MC68LC040 microprocessor running at a clock frequency of 66/33 MHz (see sidebar).
- **Upgrade path:** The user can upgrade to a PowerPC™ 603 processor by replacing the main logic board.
- **RAM:** The built-in memory consists of 4 or 8 MB of low-power, self-refreshing dynamic RAM (DRAM).
- **RAM expansion:** The computer accepts a RAM expansion card with up to 32 MB, for a total of 40 MB of RAM.
- **Display:** The computer has a built-in flat panel display, a 640-by-480 pixel LCD backlit by a cold cathode fluorescent lamp (CCFL). The display can be one of two types: 10.4-inch FSTN color or 9.5-inch supertwist grayscale.
- **Hard disk:** The computer has one internal 2.5-inch IDE hard disk drive with a capacity of 500 MB. See "Configurations" on page 5.
- **SCSI disk mode:** With an optional HDI-30 SCSI Disk Adapter cable, the computer allows the user to read and store data on the computer's internal hard disk from another Macintosh computer.
- **Expansion bay:** The computer has an opening that accepts a plug-in module with a 1.4 MB high-density disk drive, some other IDE device, or an AC power adapter.
- **PCMCIA slot:** The computer accepts one type III or two type II PCMCIA cards.
- **Modem:** The computer accepts a PCMCIA modem card or an external modem connected to the serial port.

Processor clock speeds

The MC68LC040 uses two processor clocks: one for the system bus and another, at twice the speed, for the internal circuits. Thus, an MC68LC040 with a system

bus clock of 33 MHz runs its internal processor at 66 MHz.

Introduction

- **Standard I/O ports:** The computer has all the standard Macintosh inputs and outputs, including external video output. The I/O ports are an HDI-30 connector for external SCSI devices, a 4-pin mini-DIN Apple Desktop Bus (ADB) port, an 8-pin mini-DIN serial port, a stereo audio output jack, and a video output connector for an external monitor.
- **Networking:** The computer has a built-in LocalTalk network interface.
- **Sound:** The computer has a built-in microphone and speaker as well as a stereo headphone jack.
- **Keyboard:** The computer has a full-size keyboard with function keys and power on/off control.
- **Trackpad:** The cursor-positioning device is an integrated flat pad that replaces the trackball used in previous Macintosh PowerBook computers.
- **Batteries:** The computer has space for one Macintosh PowerBook Intelligent Battery: a rechargeable battery with a built-in processor that communicates with the computer's Power Manager. The model with the color display uses a 16.8 V lithium ion battery; the grayscale model uses a nickel metal hydride battery.
- **Power supply:** The computer comes with an external recharger/power adapter that accepts any worldwide standard voltage from 100–240 VAC at 50–60 Hz.
- **Security connector:** The computer has a connector on the side panel that allows users to attach a security device. The security device also secures the battery and any module in the expansion bay.
- **Weight:** The computer weighs 6.5 pounds with the battery installed.
- **Size:** The computer is 11.3 inches wide and 8.5 inches deep. The models with grayscale displays are 2.0 inches high; models with color displays are 2.1 inches high.

Appearance

The Macintosh PowerBook 190 computer has a streamlined case that opens up like a clamshell. Figure 1-1 shows a front view of the Macintosh PowerBook 190 computer, and Figure 1-2 shows a back view.

Figure 1-1 Front view of the computer

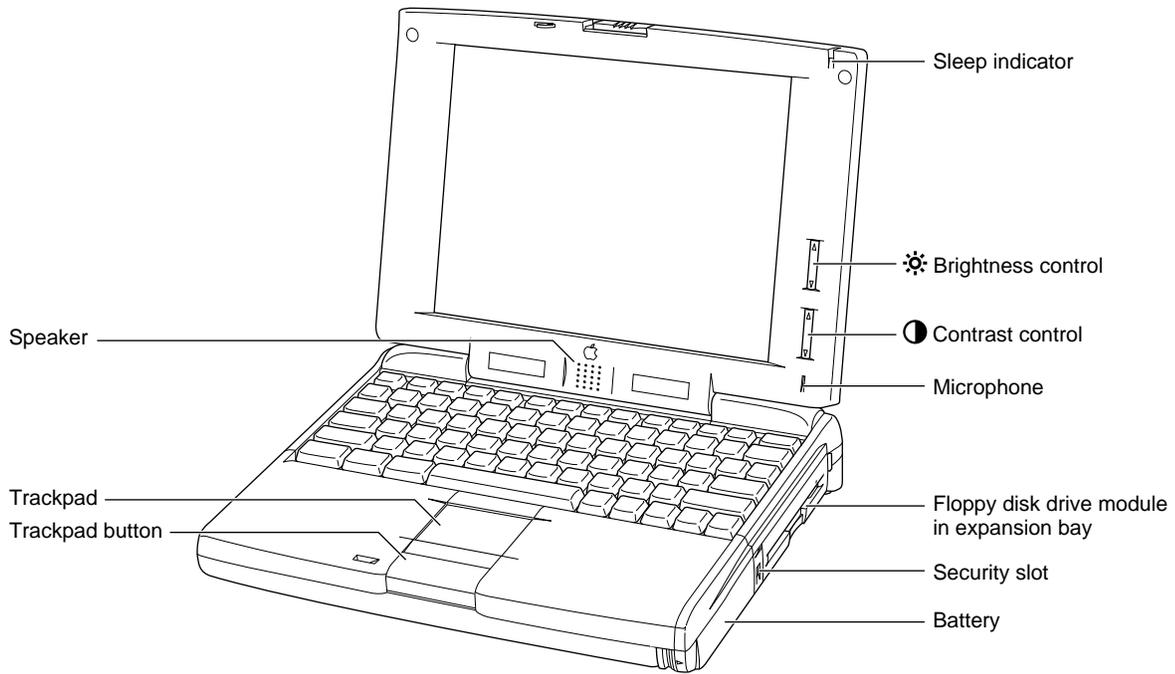
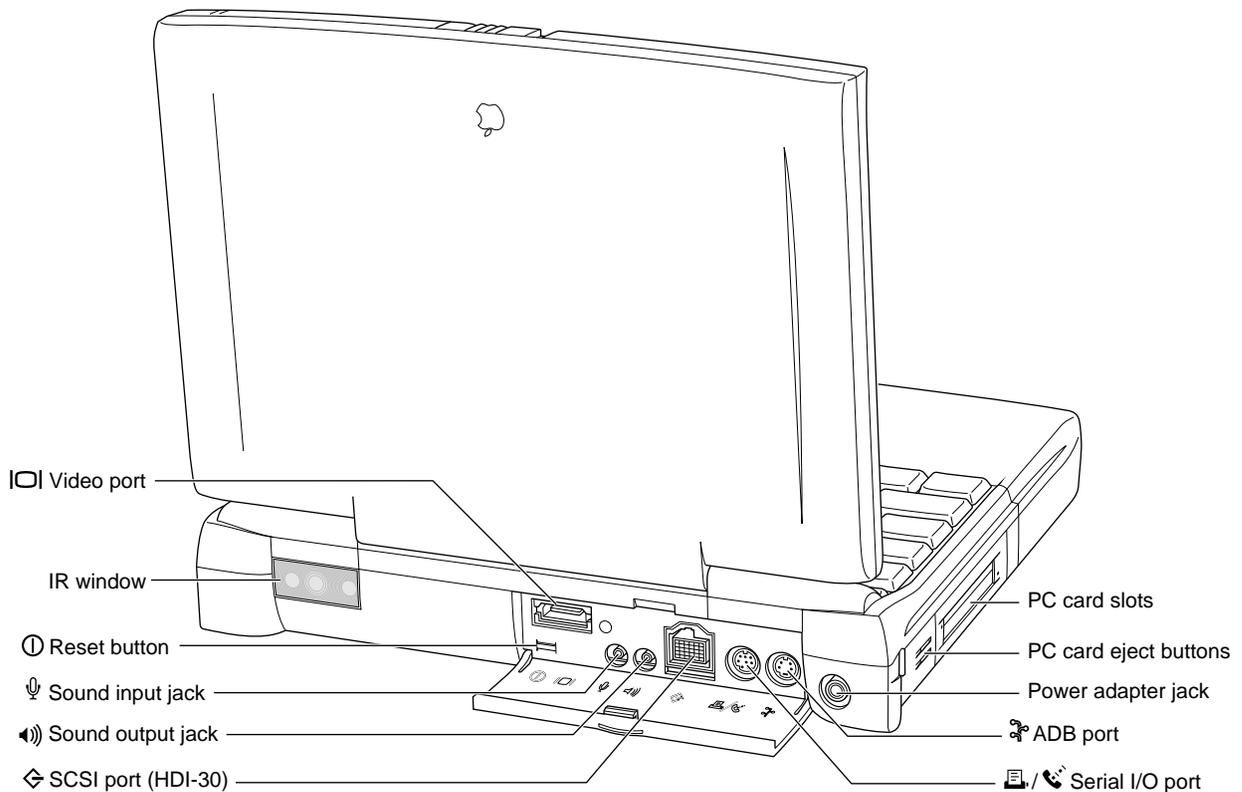


Figure 1-2 Back view of the computer

Configurations

The Macintosh Macintosh PowerBook 190 computer is available in four configurations, as shown in Table 1-1.

Table 1-1 Models and configurations

Model	RAM size	Display type	Hard disk size
Macintosh PowerBook 190	4 MB	9.5-inch supertwist grayscale	500 MB
Macintosh PowerBook 190	8 MB	9.5-inch supertwist grayscale	500 MB
Macintosh PowerBook 190cs	4 MB	10.4-inch active matrix color	500 MB
Macintosh PowerBook 190cs	8 MB	10.4-inch active matrix color	500 MB

Peripheral Devices

In addition to the devices that are included with the computer, several peripheral devices are available separately:

- The Macintosh PowerBook 8 MB Memory Expansion Kit expands the RAM to 12 or 16 MB.
- The Macintosh PowerBook 8-bit Color Video-out Upgrade Kit for the Macintosh PowerBook 190 series provides a 256-color display on an external video monitor up to 17 inches in screen size.
- The Macintosh PowerBook Infrared Upgrade Kit for the Macintosh PowerBook 190 series allows the computer to communicate with Newton PDAs and other communications devices.
- The Macintosh PowerBook Intelligent Battery is a lithium ion battery, available separately as an additional or replacement battery.
- The Macintosh PowerBook 45W AC Adapter, which comes with the computer, is also available separately. The adapter can recharge one internal battery in just four hours while the computer is running or two hours while the computer is shut down or in sleep mode.

Compatibility Issues

The Macintosh PowerBook 190 computer incorporates many significant changes from earlier Macintosh PowerBook designs. This section highlights key areas you should investigate in order to ensure that your hardware and software work properly with the new Macintosh PowerBook models. These topics are covered in more detail in subsequent sections.

RAM Expansion Cards

The RAM expansion card used in the Macintosh PowerBook 190 computer is the same new design used in the Macintosh PowerBook 5300. RAM expansion cards designed for earlier Macintosh PowerBook models will not work in the Macintosh PowerBook 190 models. See the section “RAM Expansion” beginning on page 39 for more information.

Number of Colors

The controller circuitry for the flat panel display includes a 256-entry color lookup table (CLUT) and is compatible with software that uses QuickDraw and the Palette Manager. The controller supports a palette of thousands of colors. However, due to the nature of color LCD technology, some colors are dithered or exhibit noticeable flicker. Apple has

Introduction

developed a new gamma table for the color displays that minimizes flicker and optimizes the available colors. For the active matrix color display, the effective range of the CLUT is about 260,000 colors. For the DualScan color display, the range of the CLUT is about 4000 colors.

See the section “Flat Panel Displays” beginning on page 24 for more information about the internal display hardware and LCD screen.

Video Mirror Mode

When a video card is installed and an external monitor is in use, the user can select video mirror mode, in which the external monitor mirrors (duplicates) the flat panel display. Applications that write directly to the display buffer may not be compatible with video mirror mode unless they take precautions to ensure that they do not write outside the active portion of the display. That is not a problem for applications that use QuickDraw and never write directly to the display buffer.

See the section “Video Mirroring” on page 49 for more information about video modes.

Sound Sample Rates

The Macintosh PowerBook 190 computer provides sound sample rates of 22.05 kHz, 44.1 kHz, and 48 kHz. The 22.05 kHz sample rate is slower than the 22.254 kHz sample rate used in some older Macintosh models. The 22.254 kHz sample rate was derived from the 16 MHz system clock; the 22.05 kHz rate was chosen for compatibility with the 44.1 kHz audio CD sample rate.

For sound samples made at the 22.254 kHz rate, playback at the 22.05 kHz rate is about 1 percent low in pitch. Furthermore, programs that bypass the Sound Manager and write to the sound FIFOs at the older rate now write too many samples to the FIFOs, causing some samples to be dropped. The result is a degradation in sound quality for those programs. Programs that use the Sound Manager to generate sounds are not affected by the change.

Power Manager Interface

Developers have written software that provides expanded Power Manager control for some older Macintosh PowerBook models. That software will not work in the Macintosh PowerBook 190 computer.

Until now, third-party software for the Power Manager has worked by reading and writing directly to the Power Manager’s data structures, so it has had to be updated whenever Apple brings out a new model with changes in its Power Manager software. Starting with the Macintosh PowerBook 520 and 540 computers, the system software includes interface routines for program access to the Power Manager functions, so it is no longer necessary for applications to deal directly with the Power Manager’s data structures. For more information, see *Inside Macintosh: Devices*.

Introduction

Developers should not assume that the Power Manager's data structures are the same on all Macintosh PowerBook models. In particular, developers should beware of the following assumptions regarding different PowerBook models:

- assuming that timeout values such as the hard disk spindown time reside at the same locations in parameter RAM
- assuming that the power-cycling process works the same way or uses the same parameters
- assuming that direct commands to the Power Manager microcontroller are supported on all models

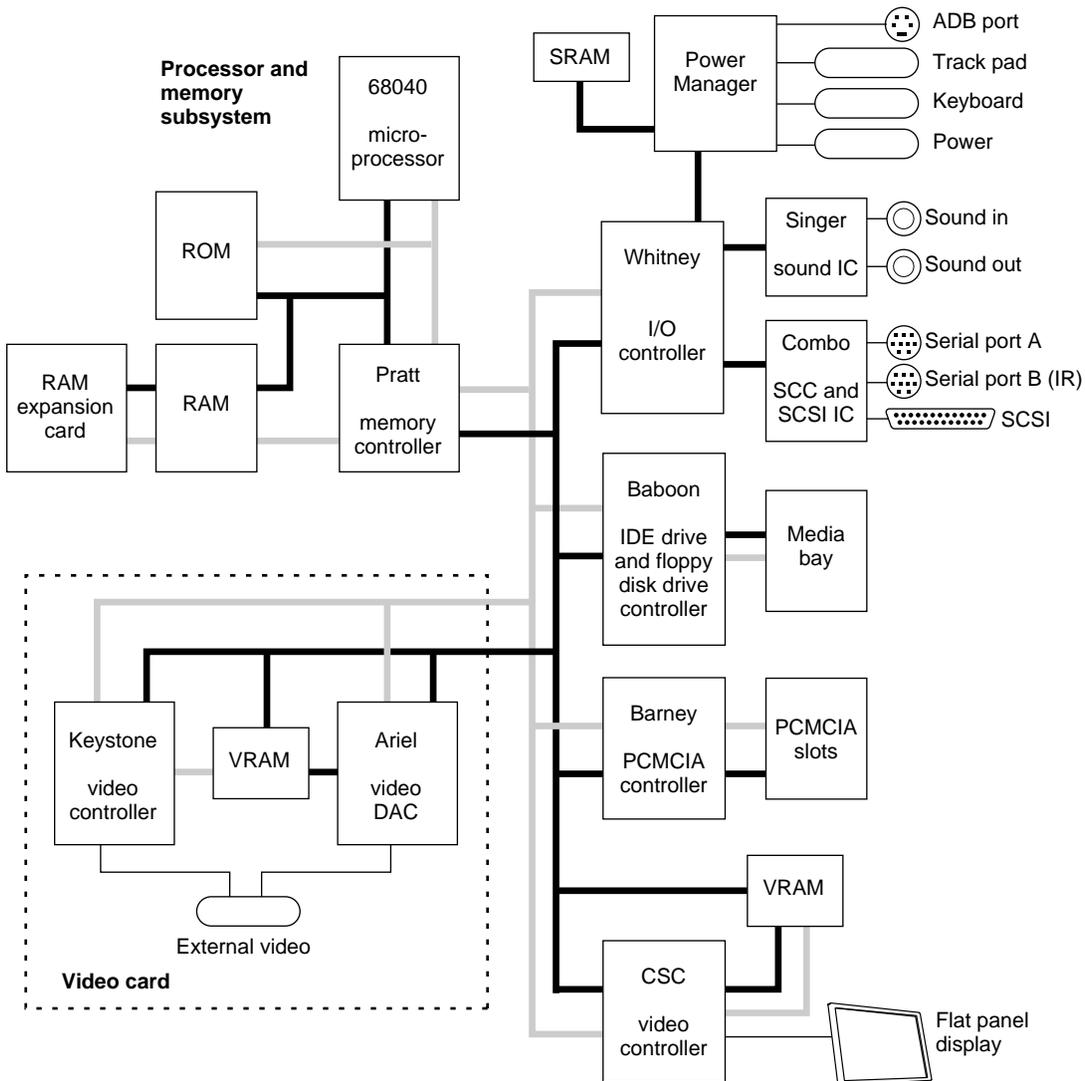
Architecture

Architecture

The architecture of the Macintosh PowerBook 190 computer is partitioned into three subsystems: the processor/memory subsystem, the input/output subsystem, and the external video card. The processor/memory subsystem operates at 33 MHz on the microprocessor bus. The input/output subsystem operates at 25 MHz on the I/O bus, a 68030-compatible bus. An Apple custom IC called the Pratt IC acts as the bridge between the two buses, translating processor bus cycles into single or multiple I/O bus cycles, as needed. The video card provides the signals for an external video monitor.

The block diagram shown in Figure 2-1 shows the two subsystems along with other modules attached to them.

Figure 2-1 Block diagram



Processor/Memory Subsystem

The processor/memory subsystem includes the MC68LC040 microprocessor, main RAM, and ROM. An optional RAM expansion card can be plugged into the logic board and becomes part of the processor/memory subsystem.

Microprocessor

The microprocessor used in the Macintosh PowerBook 190 computer is the MC68LC040. The MC68LC040 does not contain an FPU (floating-point unit). The MC68LC040 does include a built-in MMU (memory management unit).

The MC68LC040 microprocessor runs at an internal clock rate that is double its external clock rate. With an external rate of 33 MHz, the internal clock rate is 66 MHz.

For complete technical details, see the *MC68040 User's Manual* and the *MC68040 Designer's Handbook*.

RAM

The built-in RAM consists of 4 or 8 MB of dynamic RAM (DRAM). The RAM ICs are the low-power, self-refreshing type with an access time of 70 ns.

An optional RAM expansion card plugs into a 120-pin connector on the logic board. With the RAM expansion card installed, the processor/memory subsystem supports up to 40 MB of RAM. The RAM expansion card for the Macintosh PowerBook 190 computer is not compatible with the RAM card used in earlier PowerBook models. See the section "RAM Expansion" beginning on page 39 for details.

ROM

The ROM in the Macintosh PowerBook 190 computer is implemented as a 512K by 32-bit array consisting of a 1 M by 16-bit ROM IC. The ROM supports burst mode so it does not degrade the performance of the microprocessor. The ROM IC provides 2 MB of storage, which is located in the system memory map between addresses \$3000 0000 and \$3FFF FFFF. The ROM data path is 32 bits wide and addressable only as longwords. See Chapter 5, "Software Features," for a description of the features of this new ROM.

Pratt Memory Controller IC

The Pratt IC is an Apple custom IC that provides RAM and ROM memory control and also acts as the bridge between the MC68040 microprocessor bus and the MC68030 I/O bus. The Pratt IC transparently translates MC68040 bus cycles into single or multiple MC68030 dynamically sized bus cycles. Because the Pratt IC seamlessly integrates the two buses, the microprocessor and other bus masters operate as though they were on the same bus.

Architecture

The Pratt IC provides address multiplexing and refresh signals for the DRAM devices. For information about the address multiplexing, see “Address Multiplexing” on page 43. The Pratt IC supports read, write, and page mode cycles to the RAM. Pratt generates a 2048-byte CAS-before-RAS refresh cycle every 128 ms.

Bus Bridge

The Pratt IC acts as a bridge between the processor bus and the I/O bus, converting signals on one bus to the equivalent signals on the other bus. The bridge functions are performed by two converters. One accepts requests from the processor bus and presents them to the I/O bus in a manner consistent with a 68030 microprocessor. The other converter accepts requests from the I/O bus and provides access to the RAM and ROM on the processor bus.

The bus bridge in the Pratt IC runs asynchronously so that the processor bus and the I/O bus can operate at different rates. The processor bus operates at a clock rate of 33 MHz, and the I/O bus operates at 25 MHz.

Input/Output Subsystem

The input/output subsystem includes the components that communicate by way of the I/O bus:

- the Whitney custom IC
- the I/O controller ICs Combo and Singer
- the Power Manager IC
- the display controller IC
- the Baboon custom IC that controls the expansion bay
- the TREX custom IC that controls the PCMCIA slots

The next few sections describe these components.

Whitney Peripheral Support IC

The Whitney IC is a custom IC that provides the interface between the system bus and the I/O bus that supports peripheral device controllers. The Whitney IC incorporates the following circuitry:

- VIA1 like that in other Macintosh computers
- SWIM II floppy disk controller
- CPU ID register

Architecture

The Whitney IC also performs the following functions:

- bus error timing for the I/O bus
- bus arbitration for the I/O bus
- interrupt prioritization
- VIA2 functions
- sound data buffering
- clock generation
- power control signals

The Whitney IC contains the interface circuitry for the following peripheral ICs:

- Combo, which is a combination of SCC and SCSI ICs
- Singer, the sound codec IC

The Whitney IC provides the device select signals for the following ICs:

- the flat panel display controller
- the external video controller

The Whitney IC also provides the power off and reset signals to the peripheral device ICs.

Combo IC

The Combo custom IC combines the functions of the SCC IC (85C30 Serial Communications Controller) and the SCSI controller IC (53C80). The SCC portion of the Combo IC supports the serial I/O port. The SCSI controller portion of the Combo IC supports the external SCSI devices.

Singer IC

The Singer custom IC is a 16-bit digital sound codec. It conforms to the IT&T *ASCO 2300 Audio-Stereo Code Specification*. The Whitney IC maintains sound I/O buffers in main memory for sound samples being send in or out through the Singer IC. For information about the operation of the Singer IC, see the section “Sound System” on page 29.

Power Manager IC

The Power Manager IC is a 68HC05 microprocessor that operates with its own RAM and ROM. The Power Manager IC performs the following functions:

- controls sleep, shutdown, and on/off modes
- controls power to the other ICs
- controls clock signals to the other ICs
- supports the ADB

Architecture

- scans the keyboard
- controls display brightness
- monitoring battery charge level
- controls battery charging

Display Controller IC

A CSC (color support chip) IC provides the data and control interface to the LCD panel. The CSC IC contains a 256-entry CLUT, RAMDAC, display buffer controller, and flat panel control circuitry. For more information, see “Flat Panel Display Circuitry” on page 24.

Baboon Custom IC

The Baboon custom IC provides the interface to the expansion bay. The IC performs four functions:

- controls the expansion bay
- controls the IDE interfaces, both internal and in the expansion bay
- buffers the floppy disk signals to the expansion bay
- decodes addresses for the PCMCIA slots and the IDE controller

The Baboon IC controls the power to the expansion bay and the signals that allow the user to insert a device into the expansion bay while the computer is operating. Those signals are fully described in the section “Expansion Bay” beginning on page 32.

The Baboon IC controls the interface for both the internal IDE hard disk drive and a possible second IDE drive in the expansion bay. For information about the drive, see the section “Internal Hard Disk Drive” beginning on page 18. For information about the IDE drive signals in the expansion bay, see the section “Signals on the Expansion Bay Connector” beginning on page 34 and Table 4-4 on page 36.

The Baboon IC also handles the signals to a floppy disk drive installed in the expansion bay. For more information, see the section “Signals on the Expansion Bay Connector” beginning on page 34.

The address decode portion of the Baboon IC provides address decoding for the IDE controller portion of the IC. It also provides the chip select decode for the TREX custom IC and address decoding for the two PCMCIA slots.

TREX Custom IC

The TREX custom IC provides the interface and control signals for the PCMCIA slots. The main features of the TREX IC are

- the interrupt structure for the PCMCIA slots
- transfers of single-byte and word data to and from the PCMCIA slots
- power management for the PCMCIA slots, including
 - sleep mode
 - control of power to individual sockets
 - support of insertion and removal of PC cards while the computer is operating
- support for software control of card ejection
- support for time-division multiplexing (TDM), Apple Computer's technique for implementing PC cards for telecommunications

For more information about the operation of the PCMCIA slots, see "PCMCIA Slot" on page 57.

Video Card

The video card includes two additional components that communicate by way of the I/O bus:

- the Keystone custom video controller IC
- the Ariel custom video output IC

Keystone Video Controller IC

The Keystone custom IC contains the timing and control circuits for the external video circuitry. The Keystone IC has internal registers that the video driver uses to set the horizontal and vertical timing parameters. The Keystone IC also generates the video refresh addresses for the VRAM.

Ariel Video Output IC

The Ariel custom IC contains the video CLUT (color lookup table) and DAC (digital-to-analog converter). The Ariel IC takes the serial video data from the VRAM and generates the actual RGB signals for the external video monitor. The Ariel is pin and software compatible with the AC843 but does not support 24 bits per pixel.

For more information about the operation of the video card, see the section "Video Card" beginning on page 48.

I/O Features

I/O Features

This chapter describes both the built-in I/O devices and the interfaces for external I/O devices. Like the earlier chapters, it emphasizes the similarities and differences between the Macintosh PowerBook 190 computer and other PowerBook models.

This chapter describes the following built-in devices and I/O ports:

- internal IDE hard disk drive
- built-in trackpad
- built-in keyboard
- built-in flat panel display
- serial port
- SCSI port
- Apple Desktop Bus (ADB) port
- IR module
- sound system

Note

For information about the expansion bay and the optional video card, see Chapter 4, “Expansion Modules.” ♦

Internal Hard Disk Drive

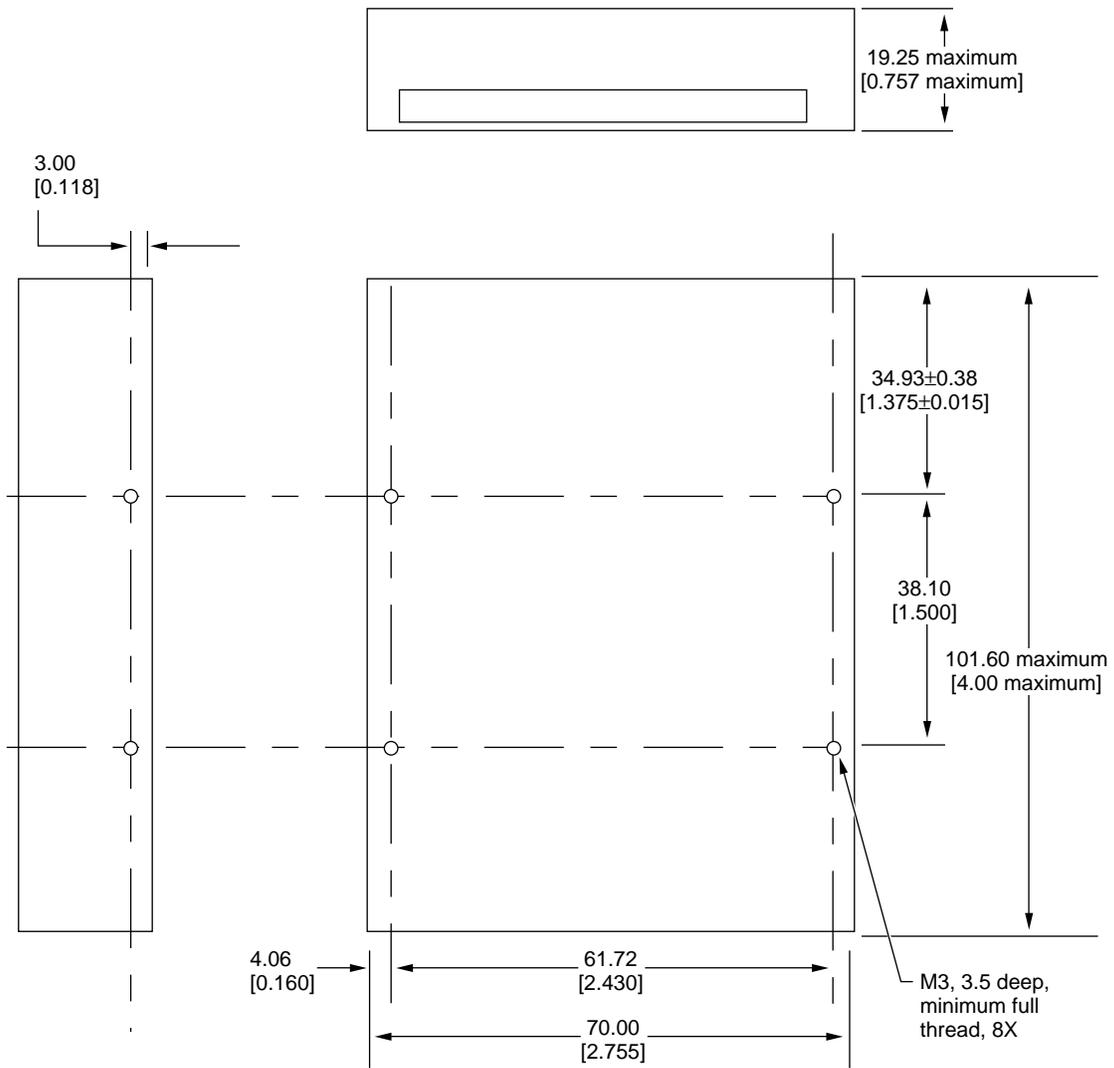
The Macintosh PowerBook 190 computer has an internal hard disk that uses the standard IDE (integrated drive electronics) interface. This interface, used for IDE drives on IBM AT-compatible computers, is also referred to as the ATA interface. The implementation of the ATA interface on the Macintosh PowerBook 190 computer is a subset of the ATA/IDE specification, ANSI proposal X3T10/0948D, Revision 2K or later (ATA-2).

For information about the software interface, see Chapter 7, “Software for ATA Devices.”

Hard Disk Specifications

Figure 3-1 shows the maximum dimensions of the hard disk and the location of the mounting holes. The minimum clearance between any conductive components on the drive and the bottom of the mounting envelope is 0.5 mm.

Figure 3-1 Maximum dimensions of the internal hard disk

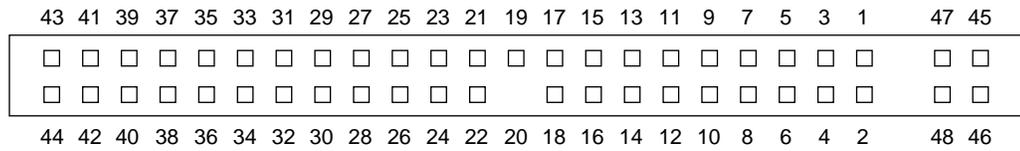


Note: Dimensions are in millimeters [inches].

Hard Disk Connectors

The internal hard disk has a 48-pin connector that carries both the IDE signals and the power for the drive. The connector has the dimensions of a 50-pin connector, but with one row of pins removed. The remaining pins are in two groups: pins 1–44, which carry the signals and power, and pins 46–48, which are reserved. Figure 3-2 shows the connector and identifies the pins. Notice that pin 20 has been removed, and that pin 1 is located nearest the gap, rather than at the end of the connector.

Figure 3-2 Connector for the internal IDE hard disk

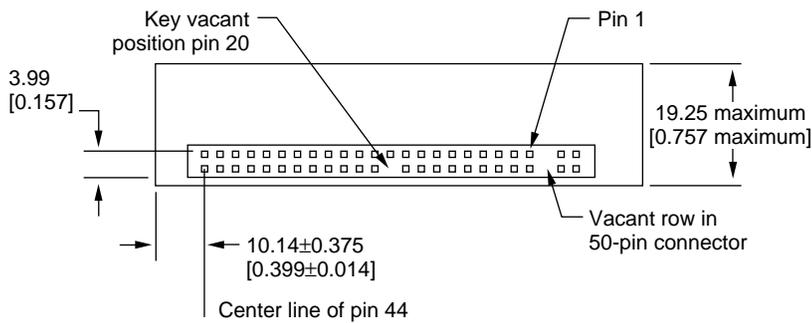


Note: gaps are equivalent to missing pins.

Connector Location

Figure 3-3 shows the position of the connector on the hard disk drive.

Figure 3-3 Position of the hard disk connector



Note: Dimensions are in millimeters [inches].

I/O Features

Pin Assignments

Table 3-1 shows the pin assignments on the 40-pin section of the hard disk connector. A slash (/) at the beginning of a signal name indicates an active-low signal.

Table 3-1 Pin assignments on the hard disk connector

Pin number	Signal name	Pin number	Signal name
1	/RESET	2	GROUND
3	DD7	4	DD8
5	DD6	6	DD9
7	DD5	8	DD10
9	DD4	10	DD11
11	DD3	12	DD12
13	DD2	14	DD13
15	DD1	16	DD14
17	DD0	18	DD15
19	GROUND	20	KEY
21	DMARQ	22	GROUND
23	/DIOW	24	GROUND
25	/DIOR	26	GROUND
27	IORDY	28	CSEL
29	/DMACK	30	GROUND
31	INTRQ	32	/IOCS16
33	DA1	34	/PDIAG
35	DA0	36	DA2
37	/CS0	38	/CS1
39	/DASP	40	GROUND
41	+5V LOGIC	42	+5V MOTOR
43	GROUND	44	Reserved

Note

The IDE data bus is connected to the I/O bus through bidirectional bus buffers. To match the big-endian format of the MC68030-compatible I/O bus, the bytes are swapped. The lower byte of the IDE data bus, DD(0–7), is connected to the high byte of the upper word of the I/O bus, IOD(24–31). The higher byte of the IDE data bus, DD(8–15), is connected to the low byte of the upper word of the I/O bus, IOD(16–23). ♦

I/O Features

IDE Signal Descriptions

Table 3-2 describes the IDE signals on the hard disk connector.

Table 3-2 Signals on the IDE hard disk connector

Signal name	Signal description
/CS0	IDE register select signal. It is asserted low to select the main task file registers. The task file registers indicate the command, the sector address, and the sector count.
/CS1	IDE register select signal. It is asserted low to select the additional control and status registers on the IDE drive.
CSEL	Cable select; if CSEL is asserted, the device address is 1; if negated, the device address is 0.
DA(0–2)	IDE device address; used by the computer to select one of the registers in the IDE drive. For more information, see the descriptions of the /CS0 and /CS1 signals.
DD(0–15)	IDE data bus; buffered from IOD(16–31) of the computer's I/O bus. DD(0–15) are used to transfer 16-bit data to and from the drive buffer. DD(8–15) are used to transfer data to and from the internal registers of the drive, with DD(0–7) driven high when writing.
/DASP	Device active or slave present.
/DIOR	IDE I/O data read strobe.
/DIOW	IDE I/O data write strobe.
/DMACK	Used by the host to initiate a DMA transfer in response to DMARQ.
DMARQ	Asserted by the device when it is ready to transfer data to or from the host.
IRDY	IDE I/O ready; when driven low by the drive, signals the CPU to insert wait states into the I/O read or write cycles.
/IOCS16	IDE I/O channel select; asserted low for an access to the data port. The computer uses this signal to indicate a 16-bit data transfer.
INTRQ	IDE interrupt request. This active-high signal is used to inform the computer that a data transfer is requested or that a command has terminated.
KEY	This pin is the key for the connector.
/PDIAG	Asserted by device 1 to indicate to device 0 that it has completed the power-on diagnostics.
/RESET	Hardware reset to the drive; an active-low signal.

Trackpad

The pointing device in the Macintosh PowerBook 190 computer is a trackpad, an integrated flat pad provides precise cursor positioning in response to motions of the user's fingertip over the surface of the pad. A single button below the trackpad is used to make selections.

The trackpad is a solid-state device that emulates a mouse by sensing the motions of the user's finger over its surface and translating those motions into ADB commands. The trackpad is lighter and more durable than the trackball used in earlier Macintosh PowerBook computers, and it consumes less power.

Also see the section "Trackpad Support" on page 65.

Keyboard

A new keyboard design provides 76 (U.S. version) or 77 (ISO version) keys, including 12 function keys. Figure 3-4 shows the version of the keyboard used on machines sold in the United States. Figure 3-5 shows the version of the keyboard used on machines sold in countries that require the ISO standard.

Figure 3-4 Keyboard, U.S. layout



Figure 3-5 Keyboard, ISO layout

By removing two screws, the user can lift out the keyboard to obtain access to the internal components and expansion connectors inside the computer.

Flat Panel Displays

The Macintosh PowerBook 190 computer has a built-in flat panel display showing 640 by 480 pixels. Two types of flat panel display are used in the different models, as shown in Table 3-3. Both types of display have a dot pitch of 0.30 mm and are backlit by a cold cathode fluorescent lamp (CCFL). Both displays can show up to 8 bits per pixel, which provides 256 colors on color displays or 256 levels of gray on grayscale displays.

Table 3-3 Characteristics of the displays

Display type	Display size (inches)	Dot pitch (mm)	Bits per pixel	Number of colors
DualScan color (FSTN)	10.4	0.30	8	256
Supertwist grayscale (FSTN)	9.5	0.27	8	256

Flat Panel Display Circuitry

The flat panel display circuitry emulates a NuBus™ video card installed in slot \$0. There is no declaration ROM as such; its functions have been incorporated into the system ROM. The display circuitry includes the CSC controller IC and a display buffer consisting of 512 KB of VRAM. The LCD display is compatible with software that uses QuickDraw and the Palette Manager. The display supports color table animation.

Types of Displays

Flat panel displays come in two types: active matrix and passive matrix.

Active matrix displays, also called thin-film transistor (TFT) displays, have a driving transistor for each individual pixel. The driving transistors give active matrix displays high contrast and fast response time.

Passive matrix refers to a display technology that does not have individual transistors. That technology is also called FSTN, for film supertwist nematic, sometimes shortened to just supertwist.

DualScan is Apple Computer's new type of FSTN color, an improved version of the color display used in the Macintosh PowerBook 165c.

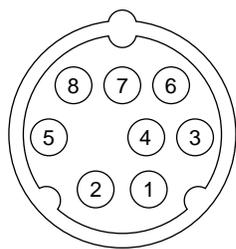
Number of Colors

The display controller IC contains a 256-entry CLUT. Although the CLUT supports a palette of thousands of colors, many of the possible colors do not look acceptable on the display. Due to the nature of color LCD technology, some colors are dithered or exhibit noticeable flicker. Apple has developed new gamma tables for these displays that minimize flicker and optimize available colors. With these gamma tables, the effective range of the CLUT for the active matrix color display is about 260,000 colors; for the DualScan color display, the effective range is about 4000 colors.

Serial Port

The Macintosh PowerBook 190 computer has a standard Macintosh serial port for synchronous, asynchronous, or AppleTalk serial communication. The 8-pin mini-DIN connector on the back panel is the same as those on other Macintosh computers. Figure 3-6 shows the connector pins and Table 3-4 shows the signal assignments.

Figure 3-6 Serial port connector



I/O Features

Table 3-4 Serial port signals

Pin number	Signal name	Signal description
1	HSKo	Handshake output
2	HSKi	Handshake input
3	TxD-	Transmit data -
4	SG	Signal ground
5	RxD-	Receive data -
6	TxD+	Transmit data +
7	GPi	General-purpose input
8	RxD+	Receive data +

SCSI Port

The SCSI port on the Macintosh PowerBook 190 computer supports the SCSI interface as defined by the American National Standards Institute (ANSI) X3T9.2 committee.

The external HDI-30 connector is identical to those used in other Macintosh PowerBook models. The SCSI portion of the Combo IC connects directly to the external SCSI connector and can sink up to 48 mA through each of the pins connected to the SCSI bus. The data and control signals on the SCSI bus are active low signals that are driven by open drain outputs.

Table 3-5 shows the signal assignments for the external SCSI connector. Note that pin 1 of the external SCSI connector is the /SCSI.DISK.MODE signal.

Table 3-5 Signals on the SCSI connector

Pin number	Signal name	Signal name
1	/SCSI.DISK.MODE	SCSI disk operating mode
2	/DB0	Bit 0 of SCSI data bus
3	GND	Ground
4	/DB1	Bit 1 of SCSI data bus
5	TERMPWR	Termination power (not used; reserved)
6	/DB2	Bit 2 of SCSI data bus
7	/DB3	Bit 3 of SCSI data bus

continued

I/O Features

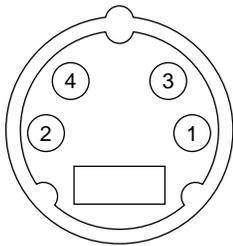
Table 3-5 Signals on the SCSI connector (continued)

Pin number	Signal name	Signal name
8	GND	Ground
9	/ACK	Acknowledge for a REQ/ACK data transfer handshake
10	GND	Ground
11	/DB4	Bit 4 of SCSI data bus
12	GND	Ground
13	GND	Ground
14	/DB5	Bit 5 of SCSI data bus
15	GND	Ground
16	/DB6	Bit 6 of SCSI data bus
17	GND	Ground
18	/DB7	Bit 7 of SCSI data bus
19	/DBP	Parity bit of SCSI data bus
20	GND	Ground
21	/REQ	Request for a REQ/ACK data transfer handshake
22	GND	Ground
23	/BSY	Indicates whether SCSI data bus is busy
24	GND	Ground
25	/ATN	Indicates an attention condition
26	/C/D	Indicates whether control or data is on the SCSI bus
27	/RST	SCSI data bus reset
28	/MSG	Indicates the message phase
29	/SEL	Selects a target or an initiator
30	/I/O	Controls the direction of data movement

ADB Port

The Apple Desktop Bus (ADB) port on the Macintosh PowerBook 190 computer is functionally the same as on other Macintosh computers. Figure 3-7 shows the pins on the connector for the ADB port.

Figure 3-7 ADB connector



The ADB is a single-master, multiple-slave serial communications bus that uses an asynchronous protocol and connects keyboards, graphics tablets, mouse devices, and other devices to the computer. The custom ADB microcontroller drives the bus and reads status from the selected external device. A 4-pin mini-DIN connector connects the ADB controller to the outside world. Table 3-6 lists the ADB connector pin assignments. For more information about the ADB, see *Guide to the Macintosh Family Hardware*, second edition.

Table 3-6 ADB connector pin assignments

Pin number	Name	Description
1	ADB	Bidirectional data bus used for input and output; an open collector signal pulled up to +5 volts through a 470-ohm resistor on the main logic board.
2	PSW	Power on signal; generates reset and interrupt key combinations.
3	+5V	+5 volts from the computer.
4	GND	Ground from the computer.

IMPORTANT

The total current available for all devices connected to the +5 V pins on the ADB is 100 mA. ▲

Infrared Module

The Macintosh PowerBook 190 computer has an infrared (IR) module that can communicate with Newton PDAs and other communications devices. When the computer is placed within a few feet of another machine with an IR interface, it can send and receive serial data using one of several standard communications protocols. The other machine may be another IR-equipped computer, a Newton PDA, or some other IR-equipped device such as a TV set.

The IR module supports the following communications protocols:

- LocalTalk
- Newton PDA
- HP-IRDA
- TV remote control

For LocalTalk operation, the IR module takes serial bits from the SCC and transmits them using a modified form of pulse encoding called PPM-4. This method of encoding uses four cycles of a 3.92 MHz carrier for each pulse, which increases the system's immunity to interference from fluorescent lights.

The modulation method used in the Newton PDA consists of gating a 500 kHz carrier on and off. This method is capable of data rates up to 38.4k bits per second.

Sound System

The 16-bit stereo audio circuitry provides high-quality sound input and output through the built-in microphone and speaker. The user can also connect external input and output devices by way of the sound input and output jacks.

The sound system is based on the Singer codec IC along with input and output amplifiers and signal conditioners. In the Macintosh PowerBook 190 computer, the Singer codec supports two channels of digital sound with sample sizes up to 16 bits and sample rates of 11 kHz, 22.05 kHz, and 44.1 kHz.

The frequency response of the sound circuits, not including the microphone and speaker, is within plus or minus 2 dB from 20 Hz to 20 kHz. Total harmonic distortion and noise is less than 0.05 percent with a 1 V rms sine wave input. The signal-to-noise ratio (SNR) is 85 dB, with no audible discrete tones.

Sound Inputs

The sound system accepts inputs from several sources:

- built-in microphone
- sound from the expansion bay
- 1-bit sound from the PCMCIA slot

The sound signal from the built-in microphone goes through a dedicated preamplifier that raises its nominal 30 mV level to the 1 V level of the codec circuits in the Singer IC.

Stereo sound signals from the expansion bay go through an analog multiplexer raises the nominal 0.5 V level of the expansion-bay sound to the 1 V input level of the codec circuits.

The sound input from the expansion bay has the following electrical characteristics:

- input impedance: 3.2k Ω
- maximum level: 0.5 V

Each PCMCIA card has one sound input pin, and the computer accepts either one or two cards. The signals from the sound input pins are mixed together and passed through a low-pass filter on their way to the codec circuits in the Singer IC. The low-pass filter has a cutoff frequency of 5 kHz. The filter has two functions: it takes the sharp edge off any tones sent from a PC card and serves as a reconstruction filter for PWM sounds generated by a PC card.

The sound input from the PCMCIA slot has the following electrical characteristics:

- input impedance: 20k Ω
- maximum level: 1 V rms

Sound Outputs

The sound system sends computer-generated sounds or sounds from the expansion bay or PC card to a built-in speaker and to an external sound output jack. The sound output jack is located on the back of the computer.

The sound output jack provides enough current to drive a pair of low-impedance headphones. The sound output jack has the following electrical characteristics:

- output impedance: 33 Ω
- minimum recommended load impedance: 32 Ω
- maximum level: 1 V rms
- maximum current: 32 mA peak

The computer turns off the sound signals to the internal speaker when an external device is connected to the sound output jack and during power cycling.

Expansion Modules

Expansion Modules

This chapter describes each of the following expansion features of the Macintosh PowerBook 190 computer:

- expansion bay
- RAM expansion
- video card (for an external monitor)
- PCMCIA slot

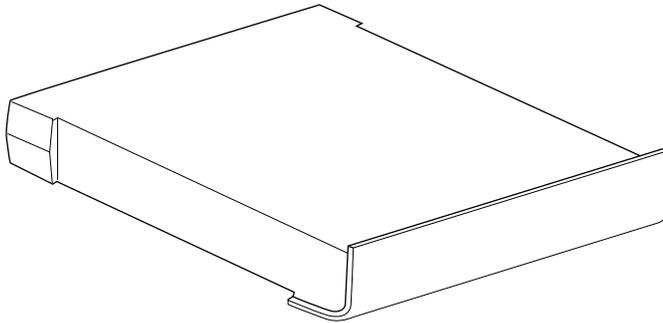
Expansion Bay

The expansion bay is an opening in the Macintosh PowerBook 190 computer that accepts a plug-in disk drive such as a floppy disk. The expansion bay can also accept a power device such as an AC adapter or a second battery.

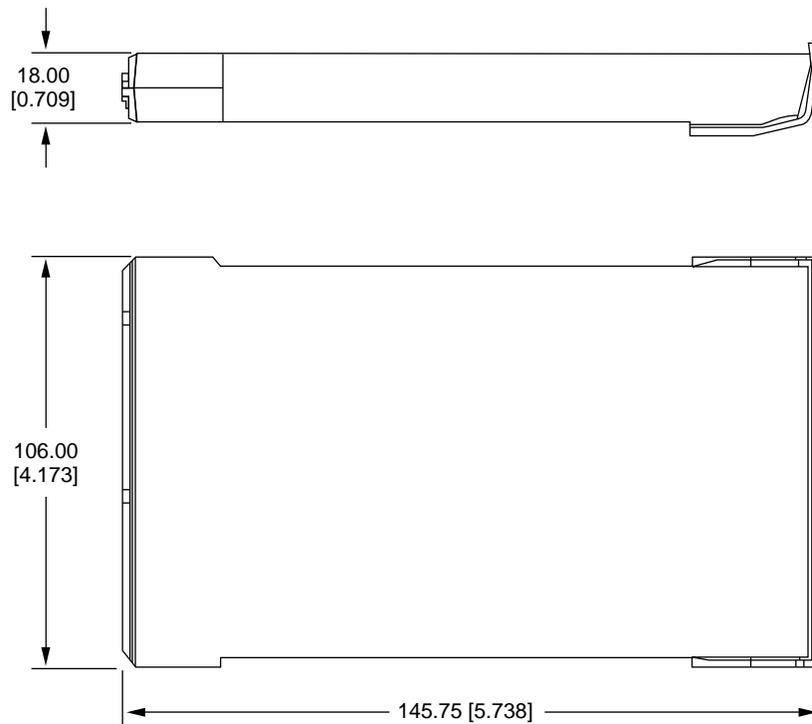
Expansion Bay Design

Figure 4-1 shows a module designed to fit into the expansion bay. Figure 4-2 shows the dimensions of the expansion bay.

Figure 4-1 Expansion bay module



Expansion Modules

Figure 4-2 Expansion bay dimensions

Note: Dimensions are in millimeters [inches].

Expansion Bay Connector

The expansion bay connector is a 90-pin shielded connector. The pins are divided into two groups by a gap. Pins 1 and 46 are at the end of the connector nearest the gap; pins 45 and 90 are at the end farthest from the gap. The connector on the main logic board is AMP part number C-93-1817-53.

A matching card connector is available as part number C-93-1817-54 from AMP, Inc. For a specification sheet or information about obtaining this connector, contact AMP at

AMP, Inc.
 19200 Stevens Creek Blvd.
 Cupertino, CA 95014-2578
 408-725-4914
 AppleLink: AMPCUPERTINO

IMPORTANT

The expansion bay connector is designed so that when a module is inserted into the expansion bay, the connections are made in the following order: first the ground by way of the connector shells, then the power pins, and last of all the signal lines. ▲

Expansion Modules

Signals on the Expansion Bay Connector

Table 4-1 shows the signal assignments on the expansion bay connector. Signal names that begin with a slash (/) are active low.

Table 4-1 Signal assignments on the expansion bay connector

Pin number	Signal name	Pin number	Signal name
1	Reserved	28	IDE_D(5)
2	Reserved	29	IDE_D(7)
3	MB_+3V	30	IDE_D(8)
4	MB_SND_COM	31	IDE_D(10)
5	Reserved	32	MB_+3V
6	Reserved	33	IDE_D(13)
7	GND	34	IDE_D(15)
8	Reserved	35	/DIOR
9	/DEV_IN	36	/CS3FX
10	DEV_ID(1)	37	Reserved
11	GND	38	IDE_ADDR(1)
12	MB_+5V	39	Reserved
13	/WRREQ	40	Reserved
14	PHASE(0)	41	Reserved
15	MB_+5V	42	Reserved
16	PHASE(3)	43	Reserved
17	WRDATA	44	Reserved
18	FD_RD	45	MB_+BAT
19	HDSEL	46	Reserved
20	GND	47	Reserved
21	Reserved	48	MB_SND_L
22	Reserved	49	MB_SND_R
23	Reserved	50	Reserved
24	IOCHRDY	51	Reserved
25	GND	52	Reserved
26	IDE_D(2)	53	Reserved
27	MB_+3V	54	DEV_ID(0)

continued

Expansion Modules

Table 4-1 Signal assignments on the expansion bay connector (continued)

Pin number	Signal name	Pin number	Signal name
55	DEV_ID(2)	73	IDE_D(6)
56	Reserved	74	GND
57	Reserved	75	IDE_D(9)
58	GND	76	IDE_D(11)
59	PHASE(1)	77	IDE_D(12)
60	PHASE(2)	78	IDE_D(14)
61	GND	79	GND
62	MB_+5V	80	/DIOW
63	/FL_ENABLE	81	/CS1FX
64	/MB_IDE_RST	82	IDE_ADDR(0)
65	Reserved	83	IDE_ADDR(2)
66	Reserved	84	GND
67	MB_+5V	85	IDE_INTRQ
68	Reserved	86	Reserved
69	IDE_D(0)	87	Reserved
70	IDE_D(1)	88	Reserved
71	IDE_D(3)	89	GND
72	IDE_D(4)	90	MB_+BAT

Signal Definitions

The signals on the expansion bay connector are of three types: expansion bay control signals, floppy disk signals, and IDE signals. The next three tables describe the three types of signals: Table 4-2 describes the control signals, Table 4-3 describes the floppy disk signals, and Table 4-4 describes the IDE signals.

Note

In Tables 4-2 through 4-4, signal names that begin with a slash (/) are active low. ♦

Expansion Modules

Table 4-2 Control signals on the expansion bay connector

Signal name	Signal description
DEV_ID(0–2)	These three signal lines identify the type of media bay device. A value of 000b identifies a floppy-disk drive; 011b identifies all other IDE devices.
/DEV_IN	This signal is low whenever a device is installed in the expansion bay; it is used by the Baboon IC to determine when a device has been inserted or removed.
MB_SND_COM	Common (ground) line for expansion bay sound signals.
MB_SND_L	Left channel sound signal from the expansion bay device.
MB_SND_R	Right channel sound signal from the expansion bay device.

Table 4-3 Floppy disk signals on the expansion bay connector

Signal name	Signal description
FD_RD	Read data from the floppy disk drive.
/FL_ENABLE	Floppy disk drive enable.
PHASE(0–3)	Phase(0–3) are state-control lines to the drive; Phase(3) is the strobe signal for writing to the drive's control registers.
WRDATA	Write data to the floppy disk drive.
/WRREQ	Write data request signal.

Table 4-4 IDE signals on the expansion bay connector

Signal name	Signal description
/CS1FX	IDE register select signal. It is asserted low to select the main task file registers. The task file registers indicate the command, the sector address, and the sector count.
/CS3FX	IDE register select signal. It is asserted low to select the additional control and status registers on the IDE drive.
/DIOR	IDE I/O data read strobe.
/DIOW	IDE I/O data write strobe.
IDE_ADDR(0–2)	IDE device address; used by the computer to select one of the registers in the IDE drive. For more information, see the descriptions of the /CS1FX and /CS3FX signals.
IDE_D(0–15)	IDE data bus, buffered from IOD(16–31) of the controller IC. IDE_D(0–15) are used to transfer 16-bit data to and from the drive buffer. IDE_D(0–7) are used to transfer data to and from the drive's internal registers, with IDE_D(8–15) driven high when writing.

continued

Expansion Modules

Table 4-4 IDE signals on the expansion bay connector (continued)

Signal name	Signal description
IOCHRDY	IDE I/O channel ready; when driven low by the IDE drive, signals the CPU to insert wait states into the I/O read or write cycles.
IDE_INTRQ	IDE interrupt request. This active high signal is used to inform the computer that a data transfer is requested or that a command has terminated.
/MB_IDE_RST	Hardware reset to the IDE drive.

Unused IDE Signals

Several signals defined in the standard interface for the IDE drive are not used by the expansion bay. Those signals are listed in Table 4-5 along with any action required for the device to operate in the media bay.

Table 4-5 Unused IDE signals

Signal name	Comment
DMARQ	No action required.
CSEL	This signal must be tied to ground to configure the device as the master in the default mode.
DMACK	This signal must be pulled high (to the IDE device's Vcc).
IOCS16	No action required.
PDIAG	No action required; the device is never operated in master-slave mode.
DAS	No action required.

Power on the Expansion Bay

Table 4-6 describes the power lines on the expansion bay connector. The MB_+5V line is controlled by the MB_PWR_EN signal from the Power Manager IC. The current drawn from MB_+5V must not exceed 1.0 A.

Table 4-6 Power for the expansion bay

Signal name	Signal description
GND	Ground.
MB_+5V	5 V power; maximum total current is 1.0 A.

User Installation of an Expansion Bay Device

The user can insert a device into the expansion bay while the computer is operating. This section describes the sequence of control events in the computer and gives guidelines for designing an expansion bay device so that such insertion does not cause damage to the device or the computer.

Sequence of Control Signals

Specific signals to the Baboon IC and the Power Manager IC allow the computer to detect the insertion of a device into the expansion bay and take appropriate action. For example, when an IDE device is inserted, the computer performs the following sequence of events:

1. When a device is inserted, the `/DEV_IN` signal goes low, causing the Baboon IC to generate an interrupt.
2. The Power Manager IC reads the three `DEV_ID` signals, which identify the device as an IDE device.
3. System software responds to the interrupt and sets a signal that turns on the power to the expansion bay.
4. When the media bay power goes high, the Baboon IC generates another interrupt.
5. System software responds to the power-on interrupt and asserts a signal to enable the IDE bus in the expansion bay.
6. The software then releases the `/MB_IDE_RST` signal from the Power Manager IC, allowing the IDE device to begin operating.

Essentially the reverse sequence occurs when a device is removed from the expansion bay:

1. When the device is removed, the `/DEV_IN` signal goes high causing the Baboon IC to generate an interrupt and disable the IDE bus.
2. System software responds to the interrupt by reading the device ID settings in the Power Manager IC, setting a signal to turn off the power to the expansion bay, and asserting the `/MB_IDE_RST` signal to disable the IDE drive.

Guidelines for Developers

Each expansion bay device must be designed to prevent damage to itself and to the computer when the user inserts or removes an expansion bay device with the computer running.

The expansion bay connector is designed so that when the device is inserted the ground and power pins make contact before the signal lines.

Even though you can design an expansion bay device that minimizes the possibility of damage when it is inserted hot—that is, while the computer is running—your instructions to the user should include warnings against doing so.

RAM Expansion

This section includes electrical and mechanical guidelines for designing a RAM expansion card for the Macintosh PowerBook 190 computer.

The RAM expansion card can contain from 8 MB to 32 MB of self-refreshing dynamic RAM in one to four banks, with 2 MB, 4 MB, or 8 MB in each bank. Table 4-7 shows how the banks can be implemented with standard RAM devices.

Table 4-7 Configurations of RAM banks

Size of bank	Number of devices per bank	Device size (bits)
2 MB	4	512K × 8
4 MB	8	1 M × 4
4 MB	2	1 M × 16
8 MB	4	2 M × 8

IMPORTANT

The RAM expansion card for the Macintosh PowerBook 190 computer is a new design; cards designed for earlier PowerBook models cannot be used in this PowerBook model. ▲

▲ WARNING

Installation of a RAM expansion card computer must be performed by an experienced technician. Installation requires care to avoid damage to the pins on the RAM expansion connector. ▲

Electrical Design Guidelines for the RAM Expansion Card

This section provides the electrical information you need to design a RAM expansion card for the Macintosh PowerBook 190 computer. The mechanical specifications are given in a subsequent section, beginning on page 46.

Connector Pin Assignments

Table 4-8 lists the names of the signals on the RAM expansion connector. Entries in the table are arranged the same way as the pins on the connector: pin 1 across from pin 2, and so on. Signal names that begin with a slash (/) are active low.

Expansion Modules

Table 4-8 Signal assignments on the RAM expansion connector

Pin	Signal name	Pin	Signal name
1	+5V_MAIN	2	+5V_MAIN
3	+3V_MAIN	4	+3V_MAIN
5	GND	6	GND
7	/RASL(2)	8	RA(11)
9	/WE	10	/RASH(2)
11	/CASL(3)	12	/CASH(3)
13	DataL(28)	14	DataH(28)
15	DataL(29)	16	DataH(29)
17	DataL(30)	18	DataH(30)
19	DataL(31)	20	DataH(31)
21	DataL(24)	22	DataH(24)
23	DataL(25)	24	DataH(25)
25	DataL(26)	26	DataH(26)
27	DataL(27)	28	DataH(27)
29	+5V_MAIN	30	+5V_MAIN
31	DataL(20)	32	DataH(20)
33	GND	34	GND
35	DataL(21)	36	DataH(21)
37	DataL(22)	38	DataH(22)
39	DataL(23)	40	DataH(23)
41	DataL(16)	42	DataH(16)
43	DataL(17)	44	DataH(17)
45	DataL(18)	46	DataH(18)
47	DataL(19)	48	DataH(19)
49	DataL(12)	50	DataH(12)
51	+3V_MAIN	52	+3V_MAIN
53	DataL(13)	54	DataH(13)
55	DataL(14)	56	DataH(14)
57	DataL(15)	58	DataH(15)
59	+5V_MAIN	60	+5V_MAIN
61	DataL(8)	62	DataH(8)

continued

Expansion Modules

Table 4-8 Signal assignments on the RAM expansion connector (continued)

Pin	Signal name	Pin	Signal name
63	GND	64	/RAM_OE
65	DataL(9)	66	DataH(9)
67	DataL(10)	68	DataH(10)
69	DataL(11)	70	DataH(11)
71	DataL(4)	72	DataH(4)
73	DataL(5)	74	DataH(5)
75	DataL(6)	76	DataH(6)
77	DataL(7)	78	DataH(7)
79	/CASH(0)	80	/RASH(1)
81	/CASH(2)	82	/CASH(1)
83	+3V_MAIN	84	+3V_MAIN
85	DataH(3)	86	DataL(3)
87	DataH(2)	88	DataL(2)
89	+5V_MAIN	90	+5V_MAIN
91	DataH(1)	92	DataL(1)
93	GND	94	GND
95	DataH(0)	96	DataL(0)
97	RA(3)	98	RA(4)
99	RA(2)	100	RA(5)
101	RA(1)	102	RA(6)
103	RA(0)	104	RA(7)
105	RA(10)	106	RA(8)
107	RA(9)	108	/RASL(0)
109	/RASL(1)	110	/RASL(3)
111	/CASL(1)	112	+12V
113	/CASL(0)	114	/RASH(0)
115	/CASL(2)	116	/RASH(3)
117	+5V_MAIN	118	+3V_MAIN
119	GND	120	GND

Expansion Modules

Signal Descriptions

Table 4-9 describes the signals on the RAM expansion connector. Signal names that begin with a slash (/) are active low.

Table 4-9 Descriptions of signals on the RAM expansion connector

Signal name	Description
+12V	12.0 V for flash memory; 30 mA maximum.
+5V_MAIN	5.0 V \pm 5%; 500 mA maximum.
+3V_MAIN	3.6 V \pm 5%; 500 mA maximum. Devices that use the +3V supply must be 5-V tolerant.
/CASH(0–3)	Column address select signals for the individual bytes in a longword. The signals are assigned to the bytes as follows: /CASH(3) selects DataH(24–31) /CASH(2) selects DataH(16–23) /CASH(1) selects DataH(8–15) /CASH(0) selects DataH(0–7)
/CASL(0–3)	Column address select signals for the individual bytes in a longword. The signals are assigned to the bytes as follows: /CASL(3) selects DataL(24–31) /CASL(2) selects DataL(16–23) /CASL(1) selects DataL(8–15) /CASL(0) selects DataL(0–7)
DataH(0–31)	Bidirectional 32-bit DRAM data bus. (DataH lines are connected to corresponding DataL lines on the main logic board.)
DataL(0–31)	Bidirectional 32-bit DRAM data bus. (DataL lines are connected to corresponding DataH lines on the main logic board.)
GND	Chassis and logic ground.
RA(0–11)	Multiplexed row and column address to the DRAM devices. (See the section “Address Multiplexing” on page 43 to determine which bits to use for a particular type of DRAM device.)
/RAM_OE	Output enable signal to the DRAM devices.
/RASL(0–3)	Row address select signals for the four banks of DRAM whose data bytes are selected by /CASL(0–3). (Signals /RASL(1–3) are for DRAM on the expansion card. The /RASL(0) signal selects a bank of DRAM on the main logic board.)
/RASH(0–3)	Row address select signals for the four banks of DRAM whose data bytes are selected by /CASH(0–3). (Signals /RASH(1–3) are for DRAM on the expansion card. The /RASH(0) signal selects a bank of DRAM on the main logic board.)
/WE	Write enable for all banks of DRAM.

Expansion Modules

In the table, signals are specified as inputs or outputs with respect to the main logic board that contains the CPU and memory module; for example, an input is driven by the expansion card into the logic board.

IMPORTANT

The last letter in the names of row and column strobe signals identifies signals that are used together: /CASL() signals are used with /RASL() signals; /CASH() signals are used with /RASH() signals. In the Macintosh PowerBook 190 computer, corresponding DataL and DataH lines are connected together. ▲

Address signals must be stable before the falling edge of RAS. Because each address line is connected to every DRAM device, whereas each RAS line is connected to only one bank of devices, the difference in loading can cause the address signals to change more slowly than the RAS signals. This situation is more likely to arise on cards with many DRAM devices. One solution is to add 100Ω damping resistors on the RAS lines.

Address Multiplexing

Signals RA(0-11) are a 12-bit multiplexed address bus and can support several different types of DRAM devices.

Depending on their internal design and size, different types of DRAM devices require different row and column address multiplexing. The operation of the multiplexing is determined by the way the address pins on the devices are connected to individual signals on the RA(0-11) bus and depends on the exact type of DRAM used.

Table 4-10 on page 44 shows how the signals on the address bus are connected for several types of DRAM devices. The device types are specified by their size and by the number of row and column address bits they require.

Table 4-10 also shows how the signals are multiplexed during the row and column address phases. For each type of DRAM device, the first and second rows show the actual address bits that drive each address pin during row addressing and column addressing, respectively. The third row shows how the device's address pins are connected to the signals on the RA(0-11) bus.

IMPORTANT

Some types of DRAM devices don't use all 12 bits in the row or column address. The table shows the bit numbers for those unused bits in italics; bit numbers for the bits that are used are shown in bold. ▲

Note

The address multiplexing scheme used in the Macintosh PowerBook 190 computer supports only the types of RAM devices shown in Table 4-10. Other RAM types should not be used. ◆

Table 4-10 Address multiplexing for some typical DRAM devices

Type of DRAM device	Individual signals on DRAM_ADDR bus											
	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
1 M × 16, 12 row bits, 8 column bits												
Row address bits	21	20	19	18	17	16	15	14	13	12	11	10
Column address bits	19	21	18	22	9	8	7	6	5	4	3	2
Device address pins	11	10	9	8	7	6	5	4	3	2	1	0
2 M × 8, 12 row bits, 9 column bits												
Row address bits	21	20	19	18	17	16	15	14	13	12	11	10
Column address bits	19	21	18	22	9	8	7	6	5	4	3	2
Device address pins	11	10	9	8	7	6	5	4	3	2	1	0
2 M × 8, 11 row bits, 10 column bits												
Row address bits	21	20	19	18	17	16	15	14	13	12	11	10
Column address bits	19	21	18	22	9	8	7	6	5	4	3	2
Device address pins	9	10	—	8	7	6	5	4	3	2	1	0

Banks of DRAM

The DRAM expansion card can have up to four banks of RAM, selected by individual signals /RASL(2–3) and /RASH(2–3). Banks can be 2 MB, 4 MB, or 8 MB in size; on a card with more than one bank, all banks must be the same size.

Because only one bank is active at a time, and because different-sized DRAM devices consume about the same amount of power when active, a card having fewer devices per bank consumes less power than a card having more devices per bank.

DRAM Device Requirements

The DRAM devices used in a DRAM expansion card must meet the following minimum specifications:

- fast page mode
- self-refreshing
- low-power grade
- row access time (t_{RAC}) of 70 ns or less
- column access time (t_{CAC}) of 20 ns or less
- page-mode cycle time (t_{PC}) of 50 ns or less

DRAM devices that use the 3-V supply must be 5-V tolerant.

Expansion Modules

Note

The DRAM refresh operation depends on the state of the computer. When the computer is operating normally, the Pratt IC provides refresh signals. When the computer goes into sleep mode, the Pratt IC switches the DRAM devices to their self-refresh feature to save power. See also “Pratt Memory Controller IC” on page 11. ♦

Expansion Card Electrical Limits

The DRAM expansion card must not exceed the following maximum current limits on the +5V supply:

Active	500 mA
Standby	24 mA
Self-refresh	6 mA

The capacitive loading on the signal lines must not exceed the following limits:

/CASL(0–3), /CASH(0–3)	40 pF
DataL(0–31), DataH(0–31)	70 pF
RA(0–11)	25 pF
/RASL(1–3), /RASH(1–3)	30 pF
/WE	85 pF

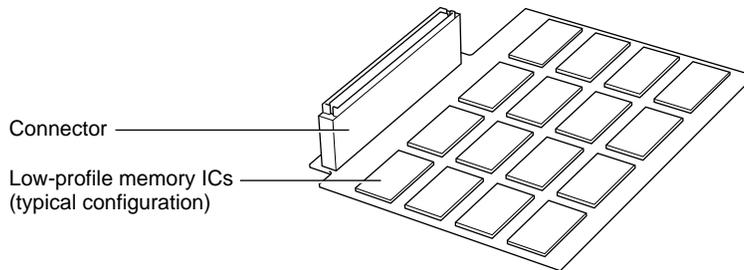
If the total capacitive loading for the devices on your card exceeds these guidelines, you should use buffers (such as 244-type devices) on the address and /RAS lines. Because of timing constraints, you cannot use buffers on the /CAS and /WE lines. If you do use buffers, you must keep within the following delay specifications:

- maximum delay on RA(): 8 ns
- maximum delay on /RASL() and /RASH(): 10 ns
- minimum delay on /RASL() and /RASH(): greater than or equal to the actual delay on RA()

Mechanical Design of the RAM Expansion Card

All the components of the RAM expansion card, including the connector, are on the same side of the card, as shown in Figure 4-3.

Figure 4-3 RAM expansion card



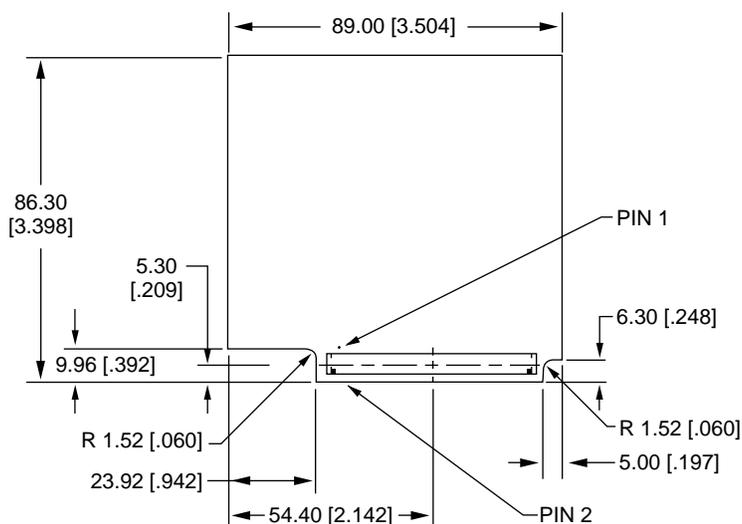
IMPORTANT

The component side is the bottom side when the card is installed. The top surface of the board must have no components or component leads. All components must reside on the bottom of the card, along with the connector. ♦

RAM Card Dimensions

Figure 4-4 is a plan view of the component side of the card showing its dimensions and the location of the connector.

Figure 4-4 Dimensions of the RAM expansion card

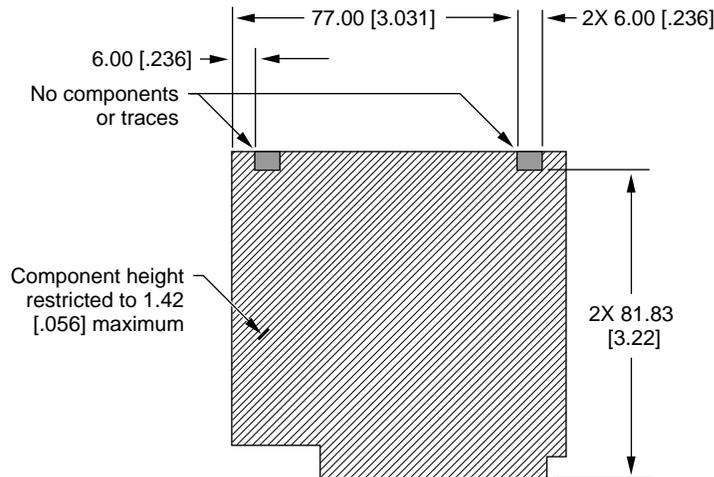


Note: Dimensions are in millimeters [inches].

Expansion Modules

Figure 4-5 shows the maximum component height and the restricted areas on the bottom (component side) of the card. Only the connector can exceed the height limit shown.

Figure 4-5 Restricted areas on the component side of the card



Note: Dimensions are in millimeters [inches].

To keep within the component height restrictions, the DRAM devices on the RAM expansion card must be of package type TSOP (thin small outline package) rather than SOP or SOJ.

IMPORTANT

The thickness of the PC board is critical; it must be within a 0.05 mm tolerance of 0.75 mm. ▲

▲ WARNING

Do not exceed the dimensions shown in the drawings. Cards that exceed these specifications may damage the computer. ▲

RAM Card Connector

The connector on the RAM expansion card is a 120-pin connector, part number KX14-120K14E9, manufactured by JAE Electronics, Irvine, California.

Note

Some early prototypes of this connector had oil contamination of the contact surfaces. Developers should avoid using those prototype connectors in their products.

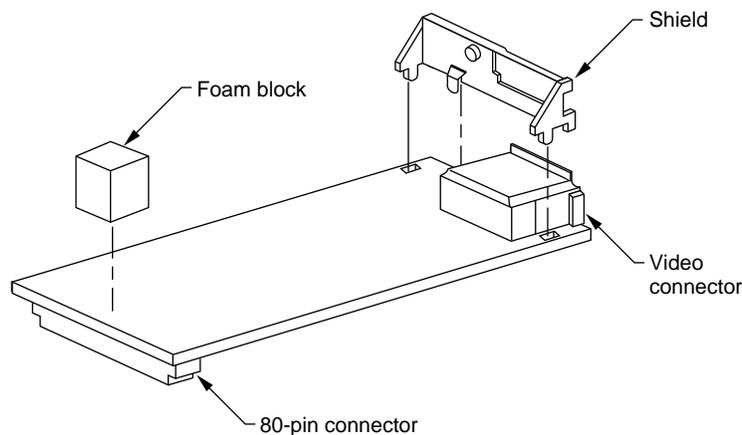
Video Card

The Macintosh PowerBook 190 computer accepts an optional video card that provides support for an external video monitor. This section describes the video card that Apple provides and includes a design guide for developers who wish to design such a card.

The Apple Video Card

Apple provides an optional video card for the Macintosh PowerBook 190 computer. Figure 4-6 shows its general appearance.

Figure 4-6 Video card



Monitors Supported

The external video card provides video output for all Apple 12-inch, 13-inch, and 16-inch RGB monitors, the Apple Macintosh Portrait Display, and Apple Computer's new 17-inch multiscan display. With appropriate adapter cables, the external video card can also support a VGA display or an 800-by-600 pixel SVGA display.

The video card contains 512 KB of video RAM, which provides pixel depths of up to 8 bits per pixel on monitor screens of up to 624 by 832 pixels.

Expansion Modules

Table 4-11 lists the video monitors supported by the video card.

Table 4-11 Video monitors and modes

Monitor type	Width (pixels)	Height (pixels)	Maximum pixel depth (bits)	Frame rate (Hz)
12-inch RGB	512	384	8	60.15
13-inch RGB*	640	480	8	66.67
Portrait	640	870	4	75.0
16-inch RGB	832	624	8	66.67
17-inch multiscan	640	480	8	66.67
17-inch multiscan	832	624	8	75.0
VGA or SVGA	640	480	8	59.95
SVGA	800	600	8	55.98

* Includes Macintosh Color Display and Apple High Resolution Monochrome Monitor.

The external video interface is enabled by attaching a monitor and restarting the computer. During the boot process, ROM software tests the monitor sense lines and activates the video output system if a recognized monitor is attached. If no monitor is found, the video output system is deactivated to conserve power.

Video Mirroring

When two video displays are used, the Macintosh PowerBook 190 computer has two video output modes: dual mode and mirror mode. In dual mode, which is the normal Macintosh mode of operation, the external video monitor is independent of the flat panel display and displays additional information. Alternatively, the user can select mirror mode, in which the external monitor mirrors (duplicates) the flat panel display.

The screen of the external monitor may be larger or smaller than the flat panel display. In mirror mode, the display on the larger screen uses only the central portion of that screen and matches the horizontal and vertical dimensions of the smaller screen.

▲ WARNING

Applications that write directly to the display buffer may not be compatible with mirror mode unless they ensure that they do not write outside the active display area. That is not a problem for applications that use QuickDraw and never write directly to the display buffer. ▲

Because the video output circuitry consumes additional power, Apple recommends that customers use the AC adapter when using an external monitor.

Expansion Modules

External Video Connector

The video card for the Macintosh PowerBook 190 computer has the same type VID-14 video output connector as the PowerBook 520 and 540 computers. An optional adapter cable allows the user to attach a standard Apple video cable. Table 4-12 lists the signal pin assignments for both the VID-14 connector on the card and the DB-15 connector on the adapter cable. Figure 4-7 shows the pin configurations of the VID-14 connector and the DB-15 connector.

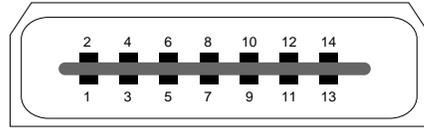
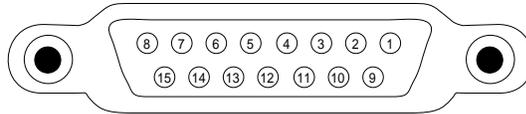
Table 4-12 Signals on the video connector

Pin		Signal name	Description
VID-14	DB-15		
1	2	RED.VID	Red video signal
2	1	RED.GND	Red video ground
3	4	SENSE0	Monitor sense signal 0
4	12	/VSYNC	Vertical synchronization signal
5	3	/CSYNC	Composite synchronization signal
6	11	GND	CSYNC and VSYNC ground
7	6	GRN.GND	Green video ground
8	5	GRN.VID	Green video signal
9	7	SENSE1	Monitor sense signal 1
10	14	HSYNC.GND	HSYNC ground
11	10	SENSE2	Monitor sense signal 2
12	15	/HSYNC	Horizontal synchronization signal
13	9	BLU.VID	Blue video signal
14	13	BLU.GND	Blue video ground
—	8	n.c.	Not connected
Shell	Shell	SGND	Shield ground

One source for the VID-14 adapter cable is

Hosiden America Corp.
 10090 Pasadena Ave., Suite B2
 Cupertino, CA 95014
 408-252-0541

Refer to Hosiden part number CMP1220-010100.

Figure 4-7 Video connectors**VID-14 connector socket****DB-15 connector socket**

Monitor Sense Codes

To identify the type of monitor connected, the video card uses the Apple monitor sense codes on the signals SENSE0–SENSE2 in Table 4-12. Table 4-13 shows the sense codes and the extended sense codes for each of the monitors the card can support. Refer to the Macintosh Technical Note *M.HW.SenseLines* for a description of the sense code system.

Table 4-13 Monitor sense codes

Monitor type	Standard	Extended sense codes		
	sense codes	(1, 2)	(0, 2)	(0, 1)
12-inch RGB	0 1 0	n.a.	n.a.	n.a.
13-inch RGB	1 1 0	n.a.	n.a.	n.a.
Portrait	0 0 1	n.a.	n.a.	n.a.
16-inch RGB	1 1 1	1 0	1 1	0 1
17-inch multiscan	1 1 0	1 1	0 1	0 0
VGA and SVGA	1 1 1	0 1	0 1	1 1
No monitor	1 1 1	1 1	1 1	1 1

Note

Both VGA and SVGA monitors have the same sense code. The first time the user starts up with an SVGA monitor, the video card treats it as a VGA monitor and shows a 640-by-480 pixel display. The user can switch to the 800-by-600 pixel SVGA mode from the Monitors control panel; when that happens, the computer changes the display to the 800-by-600 pixel display mode immediately, and continues to use that mode the next time it is started up. ♦

Video Card Design Guide

This section gives electrical and mechanical specifications for developers who wish to design a video card for the Macintosh PowerBook 190 computer.

Video Card Connector

The video card is connected to the computer's main logic board by an 80-pin connector. The connector on the card is a surface-mount connector with 0.8-mm pitch, part number KX14-80K5E9 manufactured by JAE Electronics.

Signals on the Video Card Connector

Table 4-14 shows the pin assignments on the video card connector. The table is arranged the same way as the pins on the connector, with pin 1 across from pin 2, and so on.

Table 4-14 Signals on the video card connector

Pin number	Signal name	Pin number	Signal name
1	+5V	2	+5V
3	n.c.	4	IO_DATA(8)
5	n.c.	6	GND
7	n.c.	8	IO_DATA(7)
9	IO_DATA(6)	10	IO_DATA(26)
11	IO_DATA(15)	12	IO_DATA(25)
13	IO_DATA(14)	14	IO_DATA(24)
15	IO_DATA(12)	16	IO_DATA(29)
17	IO_DATA(13)	18	IO_DATA(28)
19	IO_DATA(4)	20	IO_DATA(27)
21	GND	22	GND
23	IO_DATA(0)	24	IO_DATA(16)
25	IO_DATA(5)	26	IO_DATA(31)
27	IO_DATA(1)	28	IO_DATA(30)
29	IO_DATA(11)	30	IO_DATA(19)
31	IO_DATA(3)	32	IO_DATA(22)
33	IO_DATA(9)	34	IO_DATA(21)
35	IO_DATA(2)	36	IO_DATA(17)
37	IO_DATA(10)	38	IO_DATA(20)

continued

Expansion Modules

Table 4-14 Signals on the video card connector (continued)

Pin number	Signal name	Pin number	Signal name
39	IO_DATA(23)	40	IO_DATA(18)
41	/AS	42	IO_RW
43	/IO_RESET	44	/DSACK(1)
45	+5V	46	+5V
47	SIZ(1)	48	/DSACK(0)
49	SIZ(0)	50	IO_ADDR(0)
51	IO_ADDR(2)	52	IO_ADDR(1)
53	IO_ADDR(5)	54	IO_ADDR(3)
55	IO_ADDR(17)	56	IO_ADDR(4)
57	IO_ADDR(19)	58	IO_ADDR(7)
59	IO_ADDR(15)	60	IO_ADDR(6)
61	IO_ADDR(21)	62	IO_ADDR(10)
63	IO_ADDR(22)	64	IO_ADDR(12)
65	IO_ADDR(23)	66	IO_ADDR(13)
67	IO_ADDR(20)	68	IO_ADDR(11)
69	/KEY_CS	70	IO_ADDR(14)
71	/VID_IRQ	72	IO_ADDR(9)
73	VID_CLK	74	IO_ADDR(16)
75	+5V	76	IO_ADDR(8)
77	BUF_IOCLK	78	IO_ADDR(18)
79	GND	80	GND

Table 4-15 gives descriptions of the signals on the video card connector.

Table 4-15 Descriptions of the signals on the video card connector

Signal name	Description
/AS	Address strobe (68030 bus)
BUF_IOCLK	25 MHz I/O clock
/DSACK(0-1)	Bus data acknowledge (68030 bus)
/EXT_VID_CS	/CS for locations \$FDXX XXXX

continued

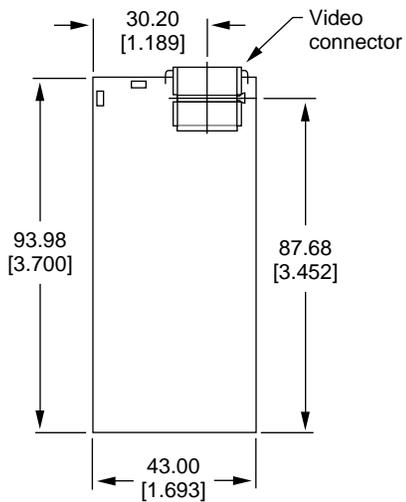
Expansion Modules

Table 4-15 Descriptions of the signals on the video card connector (continued)

Signal name	Description
IO_ADDR(0–23)	Address bus (68030 bus)
IO_DATA(0–31)	Data bus (68030 bus)
/IO_RESET	Device reset; active low
IO_RW	Read/write (68030 bus)
/KEY_CS	/CS for locations \$FEXX XXXX; reserved
SIZ(1:0)	Size of video RAM
VID_CLK	16 MHz video clock
/VID_IRQ	Video interrupt

Video Card Mechanical Design

Figure 4-8 shows the dimensions of the video card and the location of the external video connector.

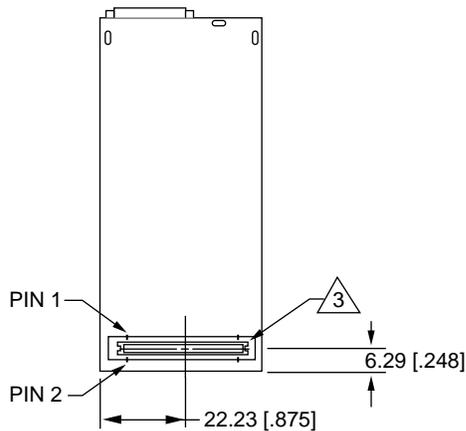
Figure 4-8 Dimensions of the video card

Note: Dimensions are in millimeters [inches].

Expansion Modules

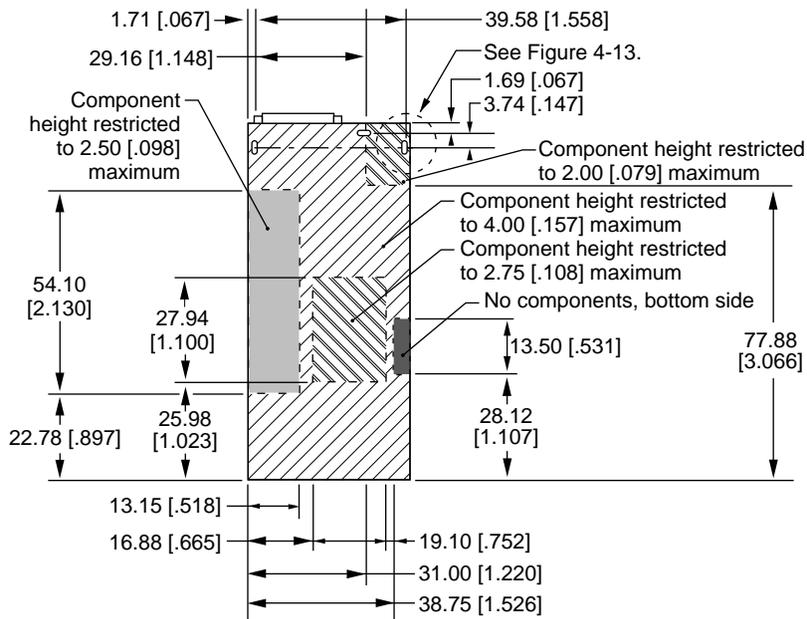
Figure 4-9 is a bottom view of the video card and shows the position of the 80-pin connector (callout 3). Figure 4-10 and Figure 4-11 show the component restrictions on the bottom and top of the card.

Figure 4-9 Video card and 80-pin connector



Note: Dimensions are in millimeters [inches].

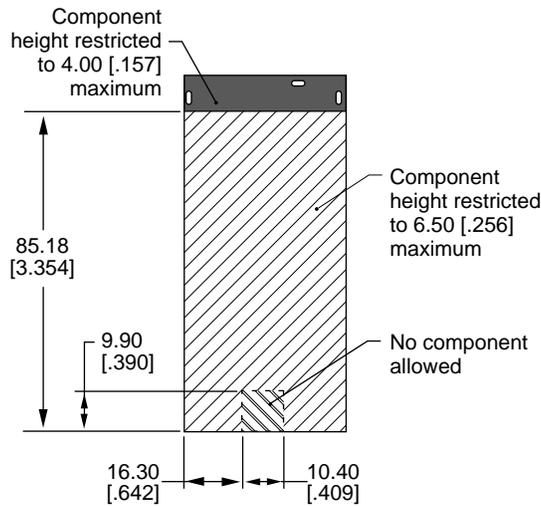
Figure 4-10 Video card bottom view with component restrictions



Note: Dimensions are in millimeters [inches].

Expansion Modules

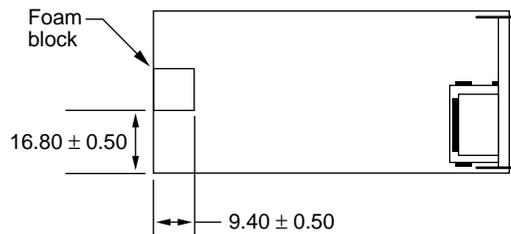
Figure 4-11 Video card top view with component restrictions



Note: Dimensions are in millimeters [inches].

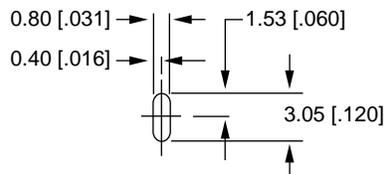
Figure 4-12 is a top view of the video card showing the position of the foam block that helps hold the card in the proper position.

Figure 4-12 Video card top view



Note: Dimensions are in millimeters [inches].

Figure 4-13 is a detail drawing showing the dimensions of the three mounting holes for the EMI shield

Figure 4-13 Detail of EMI shield mounting holes

Note: Dimensions are in millimeters [inches].

The thickness of the video card's PC board is 1.30 mm [0.051 inches].

PCMCIA Slot

The Macintosh PowerBook 190 computer has a PCMCIA slot that can accept two type II PC cards or one type III PC card. This section summarizes the features and specifications of the PCMCIA slots. For a description of the PC Card Services software, see Chapter 8, "PC Card Services." For complete specifications and descriptions of the software interfaces, developers should consult *Developing PC Card Software for the Mac OS*.

PCMCIA Features

The PCMCIA slot supports two types of PC cards: mass storage cards such as SRAM and ATA drives (both rotating hard disk and flash media) and I/O cards such as modems, network cards, and video cards. The Macintosh desktop metaphor includes the concept of storage device representation, so it already supports mass storage cards. Apple Computer has extended the metaphor to include I/O cards as well.

The user can insert or remove a PC card while the computer is operating. The user can eject a PC card either by choosing Eject from the Special menu or by dragging the card's icon to the trash.

Macintosh PowerBook computers currently support PC card ejection by software command. Software ejection is controlled by Card Services and allows Card Services to eject a PC card after notifying all clients of the card that its ejection is about to occur. If clients are using resources on the card, the clients have the option of refusing the request and alerting users to the reasons why an ejection can't take place.

Support for I/O-oriented PC cards is provided through a Macintosh Finder extension that is a client of the Card Services software. The Finder extension is responsible for maintaining card icons on the desktop, providing card information in Get Info windows, and ejecting cards when they're dragged to the Trash. The Finder extension also helps a client provide custom features such as icons, card names, card types, and help messages.

Summary Specifications

The PCMCIA slot in the Macintosh PowerBook 190 computer contains two standard PC card sockets. Each socket accepts either a type I or type II card. The PCMCIA slot also accepts one type III card, which occupies both sockets.

The mechanical and electrical characteristics of the PCMCIA slot conform to the specifications given in the *PCMCIA PC Card Standard*, Release 2.1.

The sockets support 16-bit PC cards. Each socket is 5-volt keyed and supports either a memory PC card or an I/O PC card.

Access Windows

Each socket supports the following two access windows in the computer's address space:

- one attribute memory or common memory window
- one I/O window

The only valid window combinations are the following:

- one attribute memory window
- one common memory window
- one common memory window and one I/O window

Each window has a 64 MB address space. The window address spaces could be implemented as 8 MB pages in some systems. The PCMCIA interface has the ability to map the entire PC card's memory space into the host system's memory window.

Each window has its own independent access timing register.

Data Access

Each socket supports both byte and word data access in both memory and I/O modes. The IOIS16 signal determines whether word access is single 16-bit access or two 8-bit accesses. Byte-swapping option is always big-endian mode.

The CE1 and CE2 signals determine the type of data bus access, as follows:

- word access: CE1=L, CE2=L
- even bus access: CE1=L, CE2=H
- odd bus access (not allowed): CE1=H, CE2=L

Signal Definitions

Certain signals on the PC card sockets are defined as follows:

- BVD1, BVD2: battery voltage signals (status and interrupt)
- WP: write protect (status and interrupt)
- RDY/BSY: ready/busy signal (status and interrupt)
- WAIT: used to delay access (maximum asserted time is 10 μ s)

Expansion Modules

- IRQ: interrupt request, level mode only (pulse mode is not supported)
- SPKR: speaker (digital audio output)
- STSCHG/RI: status change and ring indicator (wakeup mode)
- INPACK: not supported

Power

The PC card sockets provide power as follows:

- Vcc: programmed as either 0 V or 5 V
- Vpp1, Vpp2: programmed as either 5 V or 12 V

Vpp1 and Vpp2 cannot be programmed independently.

The maximum current from the Vcc pin is 600 mA. The maximum current from each Vpp1 or Vpp2 pin is 30 mA. The maximum current from all Vpp pins is 120 mA.

The sockets support a low-powered sleep mode.

Controller Interrupts

There is a single interrupt for both sockets. The interrupt is a combination of the status change signal and the PC card's interrupt request signal.

Software Features

Software Features

This chapter describes the new features of the software for the Macintosh PowerBook 190 computer. It describes both the built-in ROM and the system software that resides on the hard disk.

ROM Software

The ROM software in the Macintosh PowerBook 190 computer is based on the ROM used in previous PowerBook computers, with enhancements to support the many new features of these computers. Some of the features this ROM supports include the following:

- machine identification
- new memory controller IC
- Power Manager software
- new display controller
- new sound features
- ATA storage devices
- IDE disk mode
- Ethernet
- function keys
- smart batteries
- trackpad

The following sections describe each of these features.

Machine Identification

The ROM includes new tables and code for identifying the machine.

Applications can find out which computer they are running on by using the Gestalt Manager. The `gestaltMachineType` value returned by the Macintosh PowerBook 190 computer is 122 (hexadecimal \$7A). *Inside Macintosh: Overview* describes the Gestalt Manager and tells how to use the `gestaltMachineType` value to obtain the machine name string.

Software Features

Memory Controller Software

The memory control routines have been rewritten to operate with the PBX memory controller IC, which has a control register configuration different from that of the memory controller used in earlier Macintosh PowerBook models. The memory initialization and size code has been rewritten to deal with

- larger ROM size
- a new type of DRAM devices
- new memory configurations

Power Manager Software

Changes to the Power Manager software include

- support for the new lithium ion batteries
- support for turning on and off power to the Ethernet interface

Like other current PowerBook models, the Macintosh PowerBook 190 computer supports the public API for power management, which is described in *Inside Macintosh: Devices*.

Display Controller Software

The Macintosh PowerBook 190 computer has a new custom IC, the CSC (color support chip), that provides the data and control interface to the flat panel display. The ROM software includes new video drivers for that IC.

The new drivers also support a wider range of external video monitors. See “Monitors Supported” on page 48.

Sound Features

The ROM software includes new sound driver software to support the new Sound Manager, which is part of the system software. The new driver software also supports the following new features:

- improved sound performance by way of a new interface to the Singer sound IC
- support for 16-bit stereo sound input
- support for automatic gain control in software
- mixing of sound output from the modem

The new ROM software also includes routines to arbitrate control of the sound hardware between the modem and the Sound Manager.

ATA Storage Devices

Support for ATA storage devices (the internal IDE drive, PCMCIA drives, and ATAPI CD-ROM drives) is incorporated in the ROM software.

IDE Disk Mode

The ROM software also includes modifications to support disk mode. In previous Macintosh PowerBook models, the internal hard disk was a SCSI drive and the setup for disk access from another computer was called SCSI disk mode. In the Macintosh PowerBook 190 computer, the internal hard disk is an IDE drive and the disk access mode is called IDE target mode.

IDE target mode interprets SCSI commands from the external computer, translates them into the equivalent IDE commands, and calls the ATA driver to carry them out. IDE target mode does not support all SCSI commands; it supports the commands used in the Apple SCSI device driver and the new Drive Setup utility.

Note

The ATA driver is described in Chapter 8, “PC Card Services.” ♦

Ethernet Driver

The driver for the Ethernet interface can now put a sleep task for Ethernet into the Power Manager’s sleep table. This sleep task first makes a control call to the Ethernet driver to prepare the Ethernet interface IC for sleep mode. The sleep task then makes a Power Manager call to turn off power to the IC. The sleep task installs a corresponding wake task that turns the interface power back on and reinitializes the interface IC.

Support for Function Keys

The keyboard on the Macintosh PowerBook 190 computer has a row of 12 function keys across the top. Except for the function keys, the keyboard is similar to those on previous Macintosh PowerBook models. The function keys are added to the key matrix in the same way as the function keys on the Apple Extended Keyboard and return the same key codes.

Smart Battery Support

The Power Manager IC communicates with the processors in the Macintosh PowerBook Intelligent Batteries by means of a serial interface. The Power Manager’s command set has been expanded to provide system access to the data from the batteries.

Trackpad Support

The trackpad hardware, the Power Manager IC, and the system software work together to translate the movements of a finger across the surface of the trackpad into cursor movements.

The control registers for the trackpad hardware are part of the Power Manager IC. The Power Manager's software takes the raw data from the trackpad hardware and converts it to the same format as ADB mouse data before sending it on to the system software.

The ADB software that supports the trackpad includes the Cursor Device Manager, which provides a standard interface for a variety of devices. The ADB software checks to see whether a device connected to the ADB port is able to use the Cursor Device Manager. For more information, see the January 1994 revision of Technical Note HW 01, *ADB—The Untold Story: Space Aliens Ate My Mouse*.

System Software

The Macintosh PowerBook 190 computer is shipped with new system software based on System 7.5 and augmented by several new features.

IMPORTANT

Even though the software for the Macintosh PowerBook 190 computer incorporates significant changes from System 7.5, it is not a reference release: that is, it is not an upgrade for earlier Macintosh models. ▲

The system software includes changes in the following areas:

- control strip support
- support for ATA devices (IDE and ATAPI)
- large partition support
- Drive Setup, a new utility
- improved file sharing
- improved math library
- QuickDraw acceleration API
- Display Manager

These changes are described in the sections that follow.

Note

For those changes that affect the software, information about new or modified APIs is given elsewhere. Please see the cross references in the individual sections. ◆

Control Strip

The desktop on the Macintosh PowerBook 190 computer has the status and control element called the control strip that was introduced in the Macintosh PowerBook 280 and 500 models. It is a strip of graphics with small button controls and indicators in the form of various icons. For a description of the control strip and guidelines for adding modules to it, see Macintosh Technical Note *OS 06—Control Strip Modules*.

Support for ATA Devices

Support for ATA devices (the internal IDE drive, PCMCIA drives, and ATAPI CD-ROM drives) is incorporated in the ROM software.

System software for controlling the internal IDE drive and PCMCIA drives is included in a new ATA disk driver and the ATA Manager. System software for controlling the optional ATAPI CD-ROM drive is provided by a system extension in conjunction with the ATA Manager. The ATA disk driver and the ATA Manager are described in Chapter 7, “Software for ATA Devices.”

Large Partition Support

The largest disk partition supported by System 7.5 is 4 GB. The new system software extends that limit to 2 terabytes.

IMPORTANT

The largest possible file is still 2 GB. ▲

The changes necessary to support the larger partition size affect many parts of the system software. The affected software includes system-level and application-level components.

64-Bit Volume Addresses

The current disk driver API has a 32-bit volume address limitation. This limitation has been circumvented by the addition of a new 64-bit extended volume API (`PBXGetVolInfo`) and 64-bit data types (`uint64`, `XVolumeParam`, and `XIOPParam`).

For the definitions of the new API and data types, please see “The API Modifications” in Chapter 6, “Large Volume Support.”

System-Level Software

Several system components have been modified to use the 64-bit API to correctly calculate true volume sizes and read and write data to and from large disks. The modified system components are

- virtual memory code
- Disk Init
- FSM Init

Software Features

- Apple disk drivers
- HFS ROM code

Application-Level Software

Current applications do not require modification to gain access to disk space beyond the traditional 4 GB limit as long as they do not require the true size of the large partition. Applications that need to obtain the true partition size will have to be modified to use the new 64-bit API and data structures. Typical applications include utilities for disk formatting, partitioning, initialization, and backup.

The following application-level components of the system software have been modified to use the 64-bit API:

- Finder
- Finder extensions (AppleScript, AOCE Mailbox, and Catalogs)
- HDSC Setup
- Disk First Aid

In the past, the sum of the sizes of the files and folders selected in the Finder was limited to the largest value that could be stored in a 32-bit number—that is, 4 GB. By using the new 64-bit API and data structures, the Finder can now operate on selections whose total size exceeds that limit. Even with very large volumes, the Finder can display accurate information in the Folder and Get Info windows and can obtain the true volume size for calculating available space when copying.

The Finder extensions AppleScript, AOCE Mailbox, and Catalogs have been modified in the same way as the Finder because their copy-engine code is similar to that in the Finder.

A later section describes the modified Drive Setup application.

Limitations

The software modifications that support large partition sizes do not solve all the problems associated with the use of large volumes. In particular, the modifications do not address the following attributes of the file system:

- HFS file sizes are still limited to 2 GB or less.
- Large allocation block sizes cause inefficient storage. On a 2 GB volume, the minimum file size is 32 KB; on a 2-terabyte volume, the minimum file size is a whopping 32 MB.
- Drives with the new large volume driver will not mount on older Macintosh models.

Drive Setup

The software for the Macintosh PowerBook 190 computer includes a new disk setup utility named Drive Setup that replaces the old HDSC Setup utility. In addition to the

Software Features

ability to support large volumes, the Drive Setup utility has several other enhancements, including

- an improved user interface
- support for large volumes (larger than 2 GB)
- support for chainable drivers
- support for multiple HFS partitions
- the ability to mount volumes from within the Drive Setup applications
- the ability to start up (boot) from any HFS partition
- support for removable media drives

Improved File Sharing

Version 7.6 of the file-sharing software incorporates many of the features of AppleShare, including an API for servers.

The user can now set up shared files on ejectable media such as cartridge drives and CD-ROM drives. The software keeps track of the status of the shared files when the media are inserted and removed.

Math Library

The new math library (MathLib) is an enhanced version of the floating-point library included in the ROM in the first generation of Power Macintosh computers.

The new math library is bit compatible in both results and floating-point exceptions with the math library in the first-generation ROM. The only difference is in the speed of computation.

The application interface and header files for the math library have not been changed.

QuickDraw Acceleration API

The QuickDraw acceleration API is the current accelerator interface for the PowerPC version of native QuickDraw. It allows a patch chaining mechanism for decisions on categories of blit operations and also specifies the format and transport of the data to the accelerator.

Display Manager

Until now, system software has used the NuBus-specific Slot Manager to get and set information about display cards and drivers. New system software removes this explicit software dependency on the architecture of the expansion bus. The Display Manager provides a uniform API for display devices regardless of the implementation details of the devices.

Large Volume Support

Large Volume Support

This chapter describes the large volume file system for the Macintosh PowerBook 190 computer. The large volume file system is a version of the hierarchical file system (HFS) that has been modified to support volume sizes larger than the current 4 GB limit. It incorporates only the changes required to achieve that goal.

Overview of the Large Volume File System

The large volume file system includes

- modifications to the HFS ROM code, Disk First Aid, and Disk Init
- a new extended API that allows reporting of volume size information beyond the current 4 GB limit
- new device drivers and changes to the Device Manager API to support devices larger than 4 GB
- a new version of HDSC Setup that supports large volumes and chainable drivers (Chainable drivers are needed to support booting large volumes on earlier Macintosh models.)

API Changes

The system software on the Macintosh PowerBook 190 computer allows all current applications to work without modifications. Unmodified applications that call the file system still receive incorrect values for large volume sizes. The Finder and other utility programs that need to know the actual size of a volume have been modified to use the new extended `PBXGetVolInfo` function to obtain the correct value.

The existing low-level driver interface does not support I/O to a device with a range of addresses greater than 4 GB because the positioning offset (in bytes) for a read or write operation is a 32-bit value. To correct this problem, a new extended I/O parameter block record has been defined. This extended parameter block has a 64-bit positioning offset. The new parameter block and the extended `PBXGetVolInfo` function are described in “The API Modifications” beginning on page 71.

Allocation Block Size

The format of HFS volumes has not changed. What has changed is the way the HFS software handles the allocation block size. Existing HFS code treats the allocation block as a 16-bit integer. The large volume file system uses the full 32 bits of the allocation block size parameter. In addition, any software that deals directly with the allocation block size from the volume control block must now treat it as a true 32-bit value.

Even for the larger volume sizes, the number of allocation blocks is still defined by a 16-bit integer. As the volume size increases, the size of the allocation block also increases. For a 2 GB volume, the allocation block size is 32 KB, and therefore the smallest file on that disk will occupy at least 32 KB of disk space. This inefficient use of disk space is not addressed by the large volume file system.

Large Volume Support

The maximum number of files will continue to be less than 65,000. This limit is directly related to the fixed number of allocation blocks.

File Size Limits

The HFS has a maximum file size of 2 GB. The large volume file system does not remove that limit because doing so would require a more extensive change to the current API and would incur more compatibility problems.

Compatibility Requirements

The large volume file system requires at least a 68020 microprocessor or a Power Macintosh model that emulates it. In addition, the file system requires a Macintosh IIci or more recent model. On a computer that does not meet both those requirements, the large volume file system driver will not load.

The large volume file system requires System 7.5 or higher and a new Finder that supports volumes larger than 4 GB (using the new extended `PBXGetVolInfo` function).

The API Modifications

The HFS API has been modified to support volume sizes larger than 4 GB. The modifications consist of two extended data structures and a new extended `PBXGetVolInfo` function.

Data Structures

This section describes the two modified data structures used by the large volume file system:

- the extended volume parameter block
- the extended I/O parameter block

Extended Volume Parameter Block

In the current `HVolumeParam` record, volume size information is clipped at 2 GB. Because HFS volumes can now exceed 4 GB, a new extended volume parameter block is needed in order to report the larger size information. The `XVolumeParam` record contains 64-bit integers for reporting the total bytes on the volume and the number of free bytes available (parameter names `ioVTotalBytes` and `ioVFreeBytes`). In addition, several of the fields that were previously signed are now unsigned (parameter names `ioVAtrb`, `ioVBitMap`, `ioAllocPtr`, `ioValBlkSiz`, `ioVClpSiz`, `ioAlBlSt`, `ioVNxtCNID`, `ioVWrCnt`, `ioVFilCnt`, and `ioVDirCnt`).

Large Volume Support

```

struct XVolumeParam {
    ParamBlockHeader
    unsigned long    ioXVersion;        // XVolumeParam version == 0
    short           ioVolIndex;        // volume index
    unsigned long   ioVCrDate;         // date and time of creation
    unsigned long   ioVLsMod;         // date and time of last modification
    unsigned short  ioVAtrb;          // volume attributes
    unsigned short  ioVNmFls;         // number of files in root directory
    unsigned short  ioVBitMap;        // first block of volume bitmap
    unsigned short  ioAllocPtr;       // first block of next new file
    unsigned short  ioVNmAlBlks;      // number of allocation blocks
    unsigned long   ioVAlBlkSiz;      // size of allocation blocks
    unsigned long   ioVClpSiz;        // default clump size
    unsigned short  ioAlBlSt;         // first block in volume map
    unsigned long   ioVNxtCNID;       // next unused node ID
    unsigned short  ioVFrBlk;        // number of free allocation blocks
    unsigned short  ioVSigWord;       // volume signature
    short           ioVDrvInfo;        // drive number
    short           ioVDRefNum;       // driver reference number
    short           ioVFSID;          // file system identifier
    unsigned long   ioVBkUp;          // date & time of last backup
    unsigned short  ioVSeqNum;        // used internally
    unsigned long   ioVWrCnt;         // volume write count
    unsigned long   ioVFilCnt;        // number of files on volume
    unsigned long   ioVDirCnt;        // number of directories on volume
    long            ioVFndrInfo[8];    // information used by the Finder
    uint64          ioVTotalBytes;     // total number of bytes on volume
    uint64          ioVFreeBytes;     // number of free bytes on volume
};

```

Field descriptions

ioXVersion	The version of XVolumeParam; set to 0.
ioVolIndex	An index for use with the PBHGetVInfo function (described in <i>Inside Macintosh: Files</i>).
ioVCrDate	The date and time of volume initialization.
ioVLsMod	The date and time the volume information was last modified. (This field is not changed when information is written to a file and does not necessarily indicate when the volume was flushed.)
ioVAtrb	The volume attributes.
ioVNmFls	The number of files in the root directory.
ioVBitMap	The first block of the volume bitmap.
ioAllocPtr	The block at which the next new file starts. Used internally.
ioVNmAlBlks	The number of allocation blocks.

Large Volume Support

<code>ioVAlBlkSiz</code>	The size of allocation blocks.
<code>ioVClpSiz</code>	The clump size.
<code>ioAlBlSt</code>	The first block in the volume map.
<code>ioVNxtCNID</code>	The next unused catalog node ID.
<code>ioVFrBlk</code>	The number of unused allocation blocks.
<code>ioVSigWord</code>	A signature word identifying the type of volume; it's \$D2D7 for MFS volumes and \$4244 for volumes that support HFS calls.
<code>ioVDrvInfo</code>	The drive number of the drive containing the volume.
<code>ioVRefNum</code>	For online volumes, the reference number of the I/O driver for the drive identified by <code>ioVDrvInfo</code> .
<code>ioVFSID</code>	The file system identifier. It indicates which file system is servicing the volume; it's zero for File Manager volumes and nonzero for volumes handled by an external file system.
<code>ioVBkUp</code>	The date and time the volume was last backed up (it's 0 if never backed up).
<code>ioVSeqNum</code>	Used internally.
<code>ioVWrCnt</code>	The volume write count.
<code>ioVfilCnt</code>	The total number of files on the volume.
<code>ioVDirCnt</code>	The total number of directories (not including the root directory) on the volume.
<code>ioVFndrInfo</code>	Information used by the Finder.
<code>ioVTotalBytes</code>	The total number of bytes on the volume.
<code>ioVFreeBytes</code>	The number of free bytes on volume.

Extended I/O Parameter Block

The extended I/O parameter block is needed for low-level access to disk addresses beyond 4 GB. It is used exclusively by `PBRead` and `PBWrite` calls when performing I/O operations at offsets greater than 4 GB. To indicate that you are using an `XIOParam` record, you should set the `kUseWidePositioning` bit in the `ioPosMode` field.

Because file sizes are limited to 2 GB, the regular `IOParam` record should always be used when performing file-level I/O operations. The extended parameter block is intended only for Device Manager I/O operations to large block devices at offsets greater than 4 GB.

The only change from the parameter block defined on page 2-91 of *Inside Macintosh: Files* is the parameter `ioWPosOffset`, which is of type `int64`.

Note

The first eight fields are the generic HFS parameter block fields defined in *Inside Macintosh: Files*. ♦

```
struct XIOParam {
    QElemPtr    qLink;        // next queue entry
    short       qType;        // queue type
}
```

Large Volume Support

```

short      ioTrap;      // routine trap
Ptr        ioCmdAddr;   // routine address
ProcPtr    ioCompletion; // pointer to completion routine
OSErr      ioResult;   // result code
StringPtr  ioNamePtr;  // pointer to pathname
short      ioVRefNum;   // volume specification
short      ioRefNum;    // file reference number
char       ioVersNum;   // not used
char       ioPermsn;    // read/write permission
Ptr        ioMisc;     // miscellaneous
Ptr        ioBuffer;   // data buffer
unsigned long ioReqCount; // requested number of bytes
unsigned long ioActCount; // actual number of bytes
short      ioPosMode;   // positioning mode (wide mode set)
int64      ioPosOffset; // wide positioning offset
};

```

Field descriptions

<code>ioRefNum</code>	The file reference number of an open file.
<code>ioVersNum</code>	A version number. This field is no longer used; you should always set it to 0.
<code>ioPermsn</code>	The access mode.
<code>ioMisc</code>	Depending on the routine called, this field contains either a new logical end-of-file, a new version number, a pointer to an access path buffer, or a pointer to a new pathname. Because <code>ioMisc</code> is of type <code>Ptr</code> , you'll need to perform type coercion to interpret the value of <code>ioMisc</code> correctly when it contains an end-of-file (a <code>LongInt</code> value) or version number (a <code>SignedByte</code> value).
<code>ioBuffer</code>	A pointer to a data buffer into which data is written by <code>_Read</code> calls and from which data is read by <code>_Write</code> calls.
<code>ioReqCount</code>	The requested number of bytes to be read, written, or allocated.
<code>ioActCount</code>	The number of bytes actually read, written, or allocated.
<code>ioPosMode</code>	The positioning mode for setting the mark. Bits 0 and 1 of this field indicate how to position the mark; you can use the following predefined constants to set or test their value:

```
CONST
```

```

fsAtMark = 0;      {at current mark}
fsFromStart = 1;   {from beginning of file}
fsFromLEOF = 2;   {from logical end-of-file}
fsFromMark = 3;   {relative to current mark}

```

You can set bit 4 of the `ioPosMode` field to request that the data be cached, and you can set bit 5 to request that the data not be cached. You can set bit 6 to request that any data written be immediately read; this ensures that the data written to a volume exactly matches

Large Volume Support

the data in memory. To request a read-verify operation, add the following constant to the positioning mode:

```
CONST
rdVerify = 64;    {use read-verify mode}
```

You can set bit 7 to read a continuous stream of bytes, and place the ASCII code of a newline character in the high-order byte to terminate a read operation at the end of a line.

`ioPosOffset` The offset to be used in conjunction with the positioning mode.

New Extended Function

This section describes the extended `PBXGetVolInfo` function that provides volume size information for volumes greater than 4 GB.

Before using the new extended function, you should check for availability by calling the `Gestalt` function. Make your call to `Gestalt` with the `gestaltFSAttr` selector to check for new File Manager features. The response parameter has the `gestaltFSSupports2TBVolumes` bit set if the File Manager supports large volumes and the new extended function is available.

PBXGetVolInfo

You can use the `PBXGetVolInfo` function to get detailed information about a volume. It can report volume size information for volumes up to 2 terabytes.

```
pascal OSErr PBXGetVolInfo (XVolumeParam paramBlock, Boolean async);
```

`paramBlock` A pointer to an extended volume parameter block.
`async` A Boolean value that specifies asynchronous (`true`) or synchronous (`false`) execution.

An arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both:

Arrow	Meaning
→	Input
←	Output
↔	Both

Large Volume Support

Parameter block

→	ioCompletion	ProcPtr	Pointer to a completion routine.
←	ioResult	OSErr	Result code of the function.
↔	ioNamePtr	StringPtr	Pointer to the volume's name.
↔	ioVRefNum	short	On input, a volume specification; on output, the volume reference number.
→	ioXVersion	unsigned long	Version of XVolumeParam (value = 0).
→	ioVolIndex	short	Index used for indexing through all mounted volumes.
←	ioVCrDate	unsigned long	Date and time of initialization.
←	ioVLsMod	unsigned long	Date and time of last modification.
←	ioVAtrb	unsigned short	Volume attributes.
←	ioVNmFls	unsigned short	Number of files in the root directory.
←	ioVBitMap	unsigned short	First block of the volume bitmap.
←	ioVAllocPtr	unsigned short	Block where the next new file starts.
←	ioVNmAblks	unsigned short	Number of allocation blocks.
←	ioVABlkSiz	unsigned long	Size of allocation blocks.
←	ioVClpSiz	unsigned long	Default clump size.
←	ioAlBlSt	unsigned short	First block in the volume block map.
←	ioVNxtCNID	unsigned long	Next unused catalog node ID.
←	ioVFrBlk	unsigned short	Number of unused allocation blocks.
←	ioVSigWord	unsigned short	Volume signature.
←	ioVDrvInfo	short	Drive number.
←	ioVDRfNum	short	Driver reference number.
←	ioVFSID	short	File system handling this volume.
←	ioVBkUp	unsigned long	Date and time of last backup.
←	ioVSeqNum	unsigned short	Used internally.
←	ioVWrCnt	unsigned long	Volume write count.
←	ioVFilCnt	unsigned long	Number of files on the volume.
←	ioVDirCnt	unsigned long	Number of directories on the volume.

Large Volume Support

←	<code>ioVFndrInfo[8]</code>	<code>long</code>	Used by the Finder.
←	<code>ioVTotalBytes</code>	<code>uint64</code>	Total number of bytes on the volume.
←	<code>ioVFreeBytes</code>	<code>uint64</code>	Number of free bytes on the volume.

DESCRIPTION

The `PBXGetVolInfo` function returns information about the specified volume. It is similar to the `PBHGetVInfo` function described in *Inside Macintosh: Files* except that it returns additional volume space information in 64-bit integers.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBXGetVolInfo` are

Trap macro	Selector
<code>_HFSDispatch</code>	<code>\$0012</code>

RESULT CODES

<code>noErr</code>	0	Successful completion; no error occurred
<code>nsvErr</code>	-35	No such volume
<code>paramErr</code>	-50	No default volume

Software for ATA Devices

Software for ATA Devices

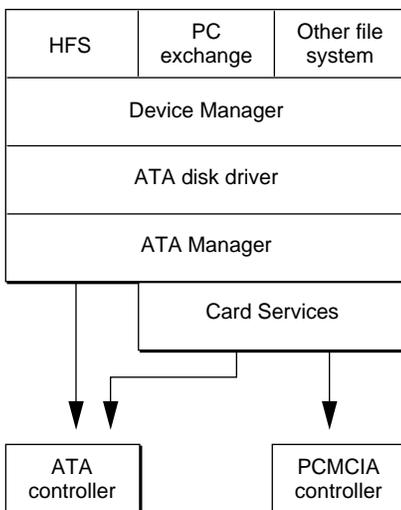
This chapter describes the system software that controls ATA devices in the Macintosh PowerBook 190 computer. To use the information in this chapter, you should already be familiar with writing programs for the Macintosh computer that call device drivers to manipulate devices directly. You should also be familiar with the ATA/IDE specification, ANSI proposal X3T10/0948D, Revision 2K or later (ATA-2).

Introduction to the ATA Software

In the Macintosh PowerBook 190 computer, the ATA software supports not only the internal ATA hard disk drive but also ATA drives installed in the expansion bay and in the PCMCIA slot. In addition to traditional Macintosh partitioned drives, the ATA software also supports other file formats such as DOS through the Macintosh PC Exchange application.

The ATA software in the Macintosh PowerBook 190 computer conforms to the Macintosh driver model. File systems communicate with the driver by way of the Device Manager, as shown in Figure 7-1. The ATA software consists of the ATA Manager and the ATA disk driver. For an ATA drive in the PCMCIA slot, the ATA software uses the Card Services software to configure the PCMCIA hardware and obtain access to the drive.

Figure 7-1 ATA software model



At the system level, the ATA disk driver and the ATA Manager work in the same way that the SCSI Manager and associated SCSI device drivers work. The ATA disk driver provides drive partition, data management, and error-handling services for the operating system as well as support for determining device capacity and controlling

Software for ATA Devices

device-specific features. The ATA Manager provides data transport services between the ATA hard disk drive and the system. The ATA Manager handles interrupts from the drives and manages the interface timing.

ATA hard disk drives appear on the desktop the same way SCSI hard disk drives currently do. Except for applications that perform low-level services such as formatting and partitioning of disk drives, applications interact with the ATA hard disk drives in a device-independent manner through the File Manager or by calling the Device Manager.

ATA Disk Driver

The ATA disk driver for the Macintosh PowerBook 190 computer has the following features:

- supports all ATA drives that comply with the ANSI ATA specification X3T10
- uses the ATA Manager for system and bus independence
- supports multiple drives and multiple partitions (volumes)
- recognizes both partitioned and non-partitioned media
- supports Macintosh PC Exchange for DOS file compatibility
- adheres to the driver rules described in *Designing PCI Cards and Drivers for Power Macintosh Computers*
- supports both synchronous and asynchronous requests from the file system
- supports manual or powered ejection of PCMCIA cards

The ATA disk driver resides in ROM and supports all ATA drives that adhere to the ANSI ATA/IDE specification X3T10. The driver can support any number of ATA drives, either internal or installed in the expansion bay or the PCMCIA slot.

The ATA disk driver relies on the services of the ATA Manager, which provides the ATA protocol engine and relieves the driver of system and bus dependencies. The main functions of the driver are managing the media and monitoring the status of the drive.

The ATA disk driver is responsible for providing block-oriented access to the storage media. The file systems treat the media as one or more logical partitions or volumes in which data at any address can be read or written indefinitely.

The ATA disk driver provides status and control functions. In addition, the driver's functionality has been augmented to support Macintosh PC Exchange and the new Drive Setup application. The functions are described in "ATA Disk Driver Reference" beginning on page 83.

The ATA disk driver supports both synchronous and asynchronous requests from the file system. The driver executes synchronous requests without relinquishing control back to the caller until completion. The driver queues asynchronous calls and returns control to the caller; it then executes the requested task in the background during interrupt time.

Drives on PC Cards

It might seem that the system should treat drives on PC cards like floppy disks because they are removable. On closer examination, the floppy disk model is not appropriate for such drives. The Mac OS assumes that a floppy disk is not partitioned and has a single HFS volume. Drives on PC cards can be quite large, making multiple partitions desirable, and they can be used in multiple platforms, so they may have formats other than HFS. For those and other reasons having to do with the way the Mac OS works, the ATA disk driver uses the hard disk storage model for PC card drives.

The hard disk model in the Mac OS assumes that the media is fixed, that is, not ejectable. The Disk Eject option in the Finder's Special menu is disabled for fixed media, but the driver can still request that an eject call be given when a volume is unmounted from the desktop (that is, when its icon is dragged to the Trash). The driver can use this eject call to eject the PC card drive when the last volume on the drive has been unmounted.

Having only the single eject call is a problem for PC card drives that have removable media because there is no way to distinguish between ejecting the media and ejecting the drive. That being the case, the ATA disk driver does not support ejection of removable media in PC card drives. It supports such drives as hard disks if the media is inserted before the drive is installed in the PCMCIA socket.

Note

The hard disk model does not permit a single drive copy. This lack should only be noticeable with single-socket systems or with a single type III drive in a stacked type II socket configuration. ♦

The PC card drive media may contain one or more individual file system partitions (volumes) displayed as icons on the desktop. The ATA disk driver mounts the volumes automatically when the PC card is inserted into a socket.

The ATA disk driver in the Macintosh PowerBook 190 computer supports both partitioned and nonpartitioned media. Partitioned media must contain a Macintosh Partition Map, or the driver treats it as nonpartitioned. The driver searches the partition map and posts disk-inserted events for all HFS, ProDOS, and other valid file system partitions it finds. If there are no valid file system partitions in the partition map or if the partition map itself does not exist, the disk driver posts a disk-inserted event for the entire media as a single partition of unknown system type. The HFS file system and installed foreign file systems such as Macintosh PC Exchange can then inspect the media to determine whether it is formatted.

Power management for PC card drives is similar to that for the internal drive, which uses an internal spindown timer to reduce power to the drive after a period of inactivity. Instead of removing power to the drive, the driver's spindown manager issues low power commands to the drive. This approach provides power conservation without incurring the performance slowdown associated with turning the drive on and off.

The driver maintains independent spindown timers for each PC card drive, allowing it to provide maximum power conservation when one or more drives is inactive. The spindown time, which can be set from the PowerBook control panel, is the same for all drives.

Software for ATA Devices

Control panels and control strip modules currently provide manual control of spindown for the internal drive by means of calls to the Power Manager. That approach doesn't work for the PC card drives. Instead, the ATA disk driver provides a new control function (`SetPowerMode`) and a new status function (`GetPowerMode`) that software can use to provide manual control of spindown.

Drives in the Expansion Bay

The ATA disk driver treats drives installed in the expansion bay the same as PC card drives except that drives in the expansion bay cannot be power ejected and the media icon on the desktop is the generic hard disk icon.

ATA Manager

The ATA Manager manages the ATA controller and its protocol. It provides data transport services between ATA devices and the system, directing commands to the appropriate device and handling interrupts from the devices.

The ATA Manager schedules I/O requests from the ATA disk driver, the operating system, and applications. The ATA Manager can handle both synchronous and asynchronous requests. When making asynchronous requests, the calling program must provide a completion routine.

The ATA Manager's internal processing of requests can be either by polling or by interrupts. When it is polling, the ATA Manager continually monitors for the next state of the protocol by looping. When it is interrupt driven, the ATA Manager is notified of the next protocol state by an interrupt. The ATA Manager determines which way to process each request as it is received; if interrupts are disabled, it processes the request by polling.

Note

The ATA Manager does not provide an access mechanism for tuples on the PCMCIA device. Any client can request tuple information from the Card Services software described in Chapter 8, "PC Card Services." ♦

The functions and data structures of the ATA Manager are described in "ATA Manager Reference" beginning on page 98.

ATA Disk Driver Reference

This section describes the routines provided by the ATA disk driver. The information in this section assumes that you are already familiar with how to use device driver routines on the Macintosh computer. If you are not familiar with Macintosh device drivers, refer to the chapter "Device Manager" in *Inside Macintosh: Devices* for additional information.

Standard Device Routines

The ATA disk driver provides the standard control and status routines described in the chapter “Device Manager” of *Inside Macintosh: Devices*. Those routines are described in this section. The specific control and status functions supported in the ATA disk driver are defined in “Control Functions” beginning on page 86 and “Status Functions” beginning on page 93.

Note

The ATA disk driver resides in ROM and is not opened or closed by applications. ♦

The Control Routine

The control routine sends control information to the ATA disk driver. The type of control function to be performed is specified in `csCode`.

The ATA disk driver implements many of the control functions supported by the SCSI hard disk device driver and defined in *Inside Macintosh: Devices* plus several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh computers*. The control functions are listed in Table 7-1 and described in “Control Functions” beginning on page 86.

Table 7-1 Control functions

Value of <code>csCode</code>	Definition
5	verify media
6	format media
7	eject drive
21	get drive icon
22	get media icon
23	get drive information
44	set startup partition
45	set partition mounting
46	set partition write protect
48	clear partition mounting
49	clear partition write protect
50	register partition
51	get a new drive
60	mount volume
70	set power-mode

RESULT CODES

noErr	Successful completion; no error occurred
controlErr	Unimplemented control call; could not complete requested operation
nsDrvErr	No such drive installed

The Status Routine

The status routine returns status information about the ATA disk driver. The type of information returned is specified in the `csCode` field, and the information itself is pointed to by the `csParamPtr` field.

The ATA disk driver implements many of the status functions supported by the SCSI hard disk device driver and defined in *Inside Macintosh: Devices*, plus several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh computers*. The status functions are listed in Table 7-2 and described in “Status Functions” beginning on page 93.

Table 7-2 Status functions

Value of <code>csCode</code>	Definition
8	drive status
43	driver gestalt
44	get boot partition
45	get partition mount status
46	get partition write protect status
51	get partition information
70	get power mode

RESULT CODES

noErr	Successful completion; no error occurred.
statusErr	Unimplemented status call; could not complete requested operation.
nsDrvErr	No such drive is installed.

Control Functions

The Control routine in the ATA disk driver supports a standard set of control functions. The functions are used for control, status, and power management.

In the function definitions, an arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both, as follows:

Arrow	Meaning
→	Input
←	Output
↔	Both

verify

The `verify` function requests a read verification of the data on the ATA hard drive media. This function performs no operation and returns `noErr` if the logical drive number is valid.

Parameter block

→	<code>csCode</code>	A value of 5.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

format

Because ATA hard drives are low-level formatted at the factory, this function does not perform any operation. The driver returns `noErr` if the logical drive number is valid.

Parameter block

→	<code>csCode</code>	A value of 6.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

eject

The `eject` function notifies the driver when a volume is no longer required by the file system. The driver performs no action unless the drive itself is ejectable (for example, a PC card drive). If the drive is ejectable and there is no other mounted volume for the drive, then the driver initiates the eject operation. When the driver is notified that the drive has been removed from the bus, the driver removes all associated logical drives from the drive queue and updates its internal records.

Parameter block

→	<code>csCode</code>	A value of 7.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.
<code>offLinErr</code>	The specified drive is not on the bus.

get drive icon

The `get drive icon` function returns a pointer to the device icon and the device name string to be displayed on the desktop when the media is initialized. If no physical icon is available the function returns the media icon. The icon is an 'ICN#' resource and varies with the system. The device name string is in Pascal format.

Parameter block

→	<code>csCode</code>	A value of 21.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>csParam[0-1]</code>	Pointer to the drive icon and name string.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

get media icon

The `get media icon` function returns a pointer to the media icon and the device name string to be displayed on the desktop for an HFS volume and in the Get Info command of the Finder. The icon is an 'ICN#' resource and varies with the type of drive or media. The device name string is in Pascal format.

Parameter block

→	<code>csCode</code>	A value of 22.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>csParam[0-1]</code>	Address of drive icon and name string (information is in ICN# format).
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

get drive information

The `get drive information` function returns information about the specified drive as defined on page 470 of *Inside Macintosh*, Volume V.

Note

This information is not in *Inside Macintosh: Devices*. ♦

Because ATA devices are not designated, all drives are designated as unspecified. Also, all drives are specified as SCSI because the only other option is IWM, which applies only to certain floppy disk drives. The internal ATA drive is specified as primary and all others as secondary. Drives on PC cards and in the expansion bay are specified as removable (meaning the drive itself, not the media).

Parameter block

→	<code>csCode</code>	A value of 23.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>csParam[0-1]</code>	Drive information value (long). \$0601 = primary, fixed, SCSI, internal. \$0201 = primary, removable, SCSI, internal.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

set startup partition

The `set startup partition` function sets the specified partition to be the startup partition. The partition is specified either by its logical drive or by its block address on the media. The current startup partition is cleared. A result code of `controlErr` is returned if the partition does not have a partition map entry on the media or if the partition could not be set to be the startup partition.

Parameter block

→	<code>csCode</code>	A value of 44.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

set partition mounting

The `set partition mounting` function enables the specified partition to be mounted. The partition is specified either by its logical drive or by its block address on the media. A result code of `controlErr` is returned if the partition does not have a partition map entry on the media or if the partition could not be enabled to be mounted.

Parameter block

→	<code>csCode</code>	A value of 45.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

set partition write protect

The `set partition write protect` function sets the specified partition to be (software) write protected. The partition is specified either by its logical drive or by its block address on the media. A result code of `controlErr` is returned if the partition does not have a partition map entry on the media or if the partition could not be set to be write protected.

Parameter block

→	<code>csCode</code>	A value of 46.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

clear partition mounting

The `clear partition mounting` function prevents the specified partition from being mounted. The partition is specified either by its logical drive or by its block address on the media. A result code of `controlErr` is returned if the partition does not have a partition map entry on the media or if the partition could not be set so as not to be mounted.

Parameter block

→	<code>csCode</code>	A value of 48.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

clear partition write protect

The `clear partition write protect` function disables the (software) write protection on the specified partition. The partition is specified either by its logical drive or by its block address on the media. A result code of `controlErr` is returned if the partition does not have a partition map entry on the media or if write protection could not be disabled.

Parameter block

→	<code>csCode</code>	A value of 49.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

register partition

The `register partition` function supports Macintosh PC Exchange. It requests the driver to redefine the starting block offset and capacity of an existing partition.

A pointer to the drive queue element is passed in along with the new physical offset and capacity. The pointer has the following form:

```
struct {
    DrvQElPte  theDrive;    // partition to be registered
    long       phyStart;    // new start offset
    long       phySize;     // new capacity (blocks)
}
```

Parameter block

→	<code>csCode</code>	A value of 50.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[0-1]</code>	Pointer to new driver information.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

get a drive

The `get a drive` function supports Macintosh PC Exchange. It requests the driver to create a new logical drive (partition) in the system drive queue. A pointer to the `DrvQElPtr` variable is passed in; this variable contains the pointer to a valid partition on the physical drive to which the new partition is to be added. Upon completion, the function returns the new `DrvQElPtr` in the variable. The `DrvQElPtr` variable is defined as follows:

```
DrvQElPtr *theDrive; //pointer to existing partition
```

Parameter block

→	<code>csCode</code>	A value of 51.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	Pointer to existing partition.
←	<code>csParam[]</code>	Pointer to new partition.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

mount volume

The `mount volume` function instructs the driver to post a disk-inserted event for the specified partition. The partition is specified either by its logical drive or by its block address on the media.

Parameter block

→	<code>csCode</code>	A value of 48.
→	<code>ioVRefNum</code>	The logical drive number, or 0 if using the partition's block address.
→	<code>csParam[0-1]</code>	The partition's block address (long) if <code>ioVRefNum</code> is 0.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>controlErr</code>	Unimplemented control call; could not complete requested operation
<code>nsDrvErr</code>	The specified logical drive number does not exist.

set power mode

The `set power mode` function changes the drive's power mode to one of four modes: active, standby, idle, and sleep. It can be used to reduce drive power consumption and decrease system noise levels.

IMPORTANT

Although the power modes have the same names as the ones in the ATA/IDE specification, they do not have the same meanings. ▲

- Active: The fully operational state with typical power consumption.
- Standby: The state with minimal power savings. The device can return to the active state in less than 5 seconds.
- Idle: The state with moderate power savings. The device can return to the active state within 15 seconds.
- Sleep: The state with minimum power consumption. The device can return to the active state within 30 seconds.

Parameter block

→	<code>csCode</code>	A value of 70.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[0]</code>	The most significant byte contains one of the following codes: 0 = enable the active mode 1 = enable the standby mode 2 = enable the idle mode 3 = enable the sleep mode (least significant byte = don't care)
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

Status Functions

The Status routine in the ATA disk driver supports a standard set of status functions. These functions are used to obtain information about a partition (volume) in an ATA hard disk drive.

drive status

The `drive status` function returns the same type of information that disk drivers are required to return for the `Status` routine, as described on page 215 of *Inside Macintosh*, Volume II.

Note

This information is not in *Inside Macintosh: Devices*. ♦

Parameter block

→	<code>csCode</code>	A value of 8.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	Not used.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

driver gestalt

The `driver gestalt` function provides the application with information about the ATA hard disk driver and the attached device. Several calls are supported under this function. A gestalt selector is used to specify a particular call.

The `DriverGestaltParam` data type defines the ATA gestalt parameter block:

```
struct DriverGestaltParam
{
    ataPBHdr                // see definition on page 99
    SInt16                  ioVRefNum;           // refNum of device
    SInt16                  csCode;            // driver gestalt code
    OSType                  driverGestaltSelector; // gestalt selector
    driverGestaltInfo driverGestaltResponse; // returned result
};
typedef struct DriverGestaltParam DriverGestaltParam;
```

Software for ATA Devices

The fields `driverGestaltSelector` and `driverGestaltResponse` are 32-bit fields.

Parameter block

→	<code>csCode</code>	A value of 43.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>driverGestaltSelector</code>	Gestalt function selector. This is a 32-bit ASCII field containing one of the following selectors: <ul style="list-style-type: none"> 'sync' Indicates synchronous or asynchronous driver. 'devt' Specifies type of device the driver is controlling. 'intf' Specifies the device interface. 'boot' Specifies PRAM value to designate this driver or device. 'vers' Specifies the version number of the driver. 'lpwr' Indicates support for low-power mode. 'dAPI' Indicates support for Macintosh PC Exchange calls. 'purg' Indicates driver can be closed or purged. 'wide' Indicates large volume support. 'ejec' Eject-call requirements.
←	<code>driverGestaltResponse</code>	Returned result based on the driver gestalt selector. The possible return values are <ul style="list-style-type: none"> 'sync' TRUE (1), indicating that the driver is synchronous. 'devt' 'disk' indicating a hard disk driver. 'intf' 'ide' for an IDE (ATA) drive, or 'pcmc' for a PC card drive. 'boot' PRAM value (long). 'vers' Current version number of the driver. 'lpwr' TRUE (1). 'dAPI' TRUE (1) 'purg' Indicates driver can be closed or purged. 'wide' TRUE (1). 'ejec' Eject call requirements (long): <ul style="list-style-type: none"> bit 0: if set, don't issue eject call on restart. bit 1: if set, don't issue eject call on shutdown.
←	<code>ioResult</code>	See result codes.

RESULT CODES

noErr	Successful completion; no error occurred.
nsDrvErr	The specified logical drive number does not exist.
statusErr	Unknown selector was specified.

get boot partition

The `get boot partition` function returns 1 if the specified partition is the boot partition, 0 if it is not. The partition is specified either by its associated logical drive or the partition's block address on the media.

Parameter block

→	csCode	A value of 44.
→	ioVRefNum	The logical drive number or 0 if using the partition's block address.
→	csParam[]	The partition's block address (long) if ioVRefNum = 0.
←	ioResult	See result codes.

RESULT CODES

noErr	Successful completion; no error occurred.
nsDrvErr	The specified logical drive number does not exist.

get partition mount status

The `get partition mount status` function returns 1 if the specified partition has mounting enabled and 0 if not enabled or if the partition does not have a partition map entry on the media. The partition is specified either by its associate logical drive or the partition's block address on the media.

Parameter block

→	csCode	A value of 45.
→	ioVRefNum	The logical drive number or 0 if using the partition's block address.
→	csParam[]	The partition's block address (long) if ioVRefNum = 0.
←	ioResult	See result codes.

RESULT CODES

noErr	Successful completion; no error occurred.
nsDrvErr	The specified logical drive number does not exist.

get partition write protect status

The `get partition write protect status` function returns 1 if the specified partition is write protected (software) and 0 if it is not. The partition is specified either by its associate logical drive or by the partition's block address on the media.

Parameter block

→	<code>csCode</code>	A value of 46.
→	<code>ioVRefNum</code>	The logical drive number or 0 if using the partition's block address.
→	<code>csParam[]</code>	The partition's block address (long) if <code>ioVRefNum = 0</code> .
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

get partition information

The `get partition information` function supports Macintosh PC Exchange. It requests the driver to return information about the partition specified by `ioVRefNum`.

The `csParam` field contains a pointer to the device information element for the return information. The pointer has the following form:

```
struct {
    DeviceIdent    SCSIID;    // device ID
                                // physical start of partition
    unsigned long physPartitionLoc;
                                // partition identifier
    unsigned long partitionNumber;
} partInfoRec, *partInfoRecPtr;
```

Parameter block

→	<code>csCode</code>	A value of 51.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	The information data structure.
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.

get power mode

The `get power mode` function returns the current power mode state of the internal hard disk. The power modes are defined on page 93.

Parameter block

→	<code>csCode</code>	A value of 70.
→	<code>ioVRefNum</code>	The logical drive number.
→	<code>csParam[]</code>	None defined.
←	<code>csParam[]</code>	The most significant byte contains one of the following codes: 0 = active mode 1 = standby mode 2 = idle mode 3 = sleep mode (least significant byte = don't care)
←	<code>ioResult</code>	See result codes.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred.
<code>nsDrvErr</code>	The specified logical drive number does not exist.
<code>statusErr</code>	The power management information couldn't be returned, due to a manager error.

ATA Manager Reference

This section defines the data structures and functions that are specific to the ATA Manager.

The ATA Manager has a single entry point through the trap \$AAF1. Functions are dispatched within the ATA Manager based on the manager function code defined in the parameter block header.

When making calls to the ATA Manager, you have to pass and retrieve parameter information through a parameter block. The size and content of the parameter block depend on the function being called. However, all calls to the ATA Manager have a common parameter block header structure. The structure of the `ataPBHdr` parameter block is common to all ATA parameter block data types. Several additional ATA parameter block data types have been defined for the various functions of the ATA Manager.

The ATA Parameter Block

This section defines the fields that are common to all ATA Manager functions that use the ATA parameter block. The fields used for specific functions are defined in the description of the functions to which they apply. You use the ATA parameter block for all calls to the ATA Manager. The `ataPBHdr` data type defines the ATA parameter block.

The parameter block includes a field, `MgrFCode`, in which you specify the function selector for the particular function to be executed; you must specify a value for this field. Each ATA function may use different fields of the ATA parameter block for parameters specific to that function.

An arrow preceding the comment indicates whether the parameter is an input parameter, an output parameter, or both, as follows:

Arrow	Meaning
→	Input
←	Output
↔	Both

The ATA parameter block header structure is defined as follows:

```

struct ataPBHdr           // ATA Manager parameter block
                          header structure
{
    Ptr      ataLink;      // reserved
    SInt16   ataQType;     // type byte
    UInt8    ataPBVers;   // → parameter block version number
    UInt8    hdrReserved; // reserved
    Ptr      hdrReserved2; // reserved
    ProcPtr  ataCompletion; // completion routine
    OSErr    ataResult;   // ← returned result
    UInt8    MgrFCode;    // → manager function code
    UInt8    ataIOSpeed;  // → I/O timing class
    UInt16   ataFlags;    // → control options
    SInt16   hdrReserved3; // reserved
    UInt32   deviceID;    // → device ID
    UInt32   TimeOut;     // → transaction timeout value
    Ptr      ataPtr1;     // client storage pointer 1
    Ptr      ataPtr2;     // client storage pointer 2
    UInt16   ataState;    // reserved, initialize to 0
    SInt16   intSemaphores; // internal semaphores
    Sint32   hdrReserved4; // reserved
};
typedef struct ataPBHdr ataPBHdr;

```

Software for ATA Devices

Field descriptions

<code>ataLink</code>	This field is reserved for use by the ATA Manager. It is used internally for queuing I/O requests. It must be initialized to 0 before calling the ATA Manager.
<code>ataQType</code>	This field is the queue type byte. It should be initialized to 0 before calling the ATA Manager.
<code>ataPBVers</code>	This field contains the parameter block version number. Values of 1 and 2 are the only values currently supported. Any other value results in a result code of <code>paramErr</code> . For individual differences between versions 1 and 2, refer to the individual functions.
<code>hdrReserved</code>	Reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.
<code>hdrReserved2</code>	Reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.
<code>ataCompletion</code>	This field contains the completion routine pointer to be called upon completion of the request. When this field is set to zero, it indicates a synchronous I/O request; a nonzero value indicates an asynchronous I/O request. The routine this field points to is called when the request has finished without error or when the request has terminated due to an error. This field is valid for any manager request. The completion routine is called as follows: <pre>pascal void (*RoutinePtr) (ataIOPB *);</pre> <p>The completion routine is called with the associated manager parameter block in the stack.</p>
<code>ataResult</code>	Completion status. This field is returned by the ATA Manager after the request has been completed. Refer to Table 7-13 on page 139 for a list of the possible error codes returned in this field.
<code>MgrFCode</code>	This field is the function selector for the ATA Manager. The functions are defined in Table 7-4 on page 104. An invalid code in this field results in an <code>ATAFuncNotSupported</code> error.
<code>ataIOSpeed</code>	This field specifies the I/O cycle timing requirement of the specified ATA drive. This field should contain word 51 of the drive identification data. Currently values 0 through 3 are supported, as defined in the ATA/IDE specification. See the ATA/IDE specification for the definitions of the timing values. If a timing value higher than one supported is specified, the manager operates in the fastest timing mode supported by the manager. Until the timing value is determined by examining the drive identification data returned by the <code>ATA_Identify</code> function, the client should request operations using the slowest mode (mode 0).
<code>ataFlags</code>	This 16-bit field contains control settings that indicate special handling of the requested function. The control bits are defined in Table 7-3 on page 101.
<code>hdrReserved3</code>	Reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.

Software for ATA Devices

`deviceID` A short word that uniquely identifies an ATA device. The field consists of the following structure:

```
struct deviceIdentification
{
  UInt16 Reserved;    // the upper word is reserved
  UInt16 deviceNum;  // consists of device ID and bus ID
};
typedef struct deviceIdentification
        deviceIdentification;
```

Bit 15 of the `deviceNum` field indicates master (=0) /slave (=1) selection. Bits 14 through 0 contain the bus ID (for example, \$0 = master unit of bus 0, \$80 = slave unit of bus 0). The present implementation allows only one device in the master configuration. This value is always 0.

`TimeOut` This field specifies the transaction timeout value in milliseconds. A value of 0 disables the transaction timeout detection.

`ataPtr1` This pointer field is available for application use. It is not modified by the ATA Manager.

`ataPtr2` This pointer field is available for application use. It is not modified by the ATA Manager.

`ataState` This field is used by the ATA Manager to keep track of the current bus state. This field must contain 0 when calling the ATA Manager.

`intSemaphores` This field is used internally by the ATA Manager. It should be set to 0 before calling the ATA Manager.

`hdrReserved4` Reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.

Table 7-3 describes the functions of the control bits in the `ataFlags` field.

Table 7-3 Control bits in the `ataFlags` field

Name	Bit	Definition
LED Enable	0	Some systems are equipped with an activity LED controlled by software. Setting this bit to 1 indicates that the LED should be turned on for this transaction. The LED is automatically turned off at the end of the transaction. Setting the bit to 0 indicates that the LED should not be turned on for this transaction. This bit has no effect in systems with no activity LED.
—	1–2	Reserved.

continued

Table 7-3 Control bits in the `ataFlags` field (continued)

Name	Bit	Definition
<code>RegUpdate</code>	3	<p>When set to 1, this bit indicates that a set of device registers should be reported back upon completion of the request. This bit is valid for the <code>ATA_ExecIO</code> function only. Refer to the description on page 113 for details. The following device registers are reported back:</p> <ul style="list-style-type: none"> sector count register sector number register cylinder register(s) SDH register
<code>ProtocolType</code>	4–5	<p>These 2 bits specify the type of command. The following command types are defined:</p> <ul style="list-style-type: none"> \$0 = standard ATA \$1 = reserved \$2 = ATAPI <p>These bits are used to indicate special protocol handling.</p> <p>For ATA command values of \$A0 or \$A1, this field must contain the ATAPI setting. For all other ATA commands, this field must contain the standard ATA setting.</p>
—	6–7	Reserved.
<code>SGType</code>	8–9	<p>This 2-bit field specifies the type of scatter gather list passed in. This field is only valid for read/write operations.</p> <p>The following types are defined:</p> <ul style="list-style-type: none"> 00 = scatter gather disabled 01 = scatter gather type I enabled 10 = reserved 11 = reserved <p>When set to 0, this field indicates that the <code>ioBuffer</code> field contains the host buffer address for this transfer, and the <code>ioReqCount</code> field contains the byte transfer count.</p> <p>When set to 1, this field indicates that the <code>ioBuffer</code> and the <code>ioReqCount</code> fields of the parameter block for this request point to a host scatter-gather list and the number of scatter-gather entries in the list, respectively.</p>

continued

Table 7-3 Control bits in the `ataFlags` field (continued)

Name	Bit	Definition
		<p>The format of the scatter-gather list is a series of the following structure definition:</p> <pre> struct IOBlock // SG entry structure { UInt8* ioBuffer; // → data buffer pointer UInt32 ioReqCount; // → byte count }; typedef struct IOBlock IOBlock; </pre>
QLockOnError	10	<p>When set to 0, this bit indicates that an error during the transaction should not freeze the I/O queue for the device. When an error occurs on an I/O request with this bit set to 0, the next queued request is processed without interruption. If an error occurs when this bit is set, however, any subsequent request without the Immediate bit set is held off until an I/O Queue Release command is received. This allows the ATA Manager to preserve the error state so that a client can examine it.</p> <p>When this bit is set, only those requests with the Immediate bit set are processed. Use this bit with caution; it can cause the system to hang if not handled correctly.</p>
Immediate	11	<p>When this bit is set to 1, it indicates that the request must be executed as soon as possible and that the status of the request must be returned. It forces the request to the head of the I/O queue for immediate execution. When this bit is set to 0, the request is queued in the order it is received and is executed according to that order.</p>
ATAioDirection	12–13	<p>This bit field specifies the direction of data transfer. Bit values are binary and are defined as follows:</p> <ul style="list-style-type: none"> 00 = no data transfer 10 = data direction in (read) 01 = data direction out (write) 11 = reserved <p>Note: These bits do not need to be set to reflect the direction of the command packet bytes.</p>

continued

Table 7-3 Control bits in the `ataFlags` field (continued)

Name	Bit	Definition
—	14	Reserved.
ByteSwap	15	When set to 1, this bit indicates that every byte of data prior to transmission on write operations and upon reception on read operations is to be swapped. When this bit is set to 0, it forces bytes to go out in the LSB-MSB format that is compatible with IBM clones. Typically, this bit should be set to 0. Setting this bit has performance implications because the byte swap is performed by the software. Use this bit with caution. Caution: Setting this bit to 1 causes the bytes in ATAPI command packets to be swapped.

Functions

This section describes the ATA Manager functions that are used to manage and perform data transfers. Each function is requested through a parameter block specific to that service. A request for an ATA function is specified by a function code within the parameter block. The entry point for all the functions is the same.

The function names and ATA Manager function codes are shown in Table 7-4.

Table 7-4 ATA Manager functions

Function name	Code	Description
ATA_Abort	\$10	Terminate the command.
ATA_BusInquiry	\$03	Get bus information.
ATA_DrvrRegister	\$85	Register the driver reference number.
ATA_DrvrDeregister	\$87	Deregister the driver reference number.
ATA_EjectDrive	\$89	Auto-eject the drive.
ATA_ExecIO	\$01	Execute ATA I/O.
ATA_FindRefNum	\$86	Look up the driver reference number.
ATA_GetDevConfig	\$8A	Get the device configuration.
ATA_GetDevLocationIcon	\$8C	Get the device location icon and string.
ATA_Identify	\$13	Get the drive identification data.
ATA_MgrInquiry	\$90	Get information about the ATA Manager and the system configuration.
ATA_ModifyDrvrEventMask	\$88	Modify the driver event mask.

continued

Table 7-4 ATA Manager functions (continued)

Function name	Code	Description
ATA_NOP	\$00	Perform no operation.
ATA_QRelease	\$04	Release the I/O queue.
ATA_RegAccess	\$12	Obtain access to an ATA device register.
ATA_ResetBus	\$11	Reset the ATA bus.
ATA_SetDevConfig	\$8B	Set the device configuration.

ATA_Abort

You can use the `ATA_Abort` function to terminate a queued I/O request. This function applies to asynchronous I/O requests only. The `ATA_Abort` function searches through the I/O queue associated with the selected device and aborts the matching I/O request. The current implementation does not abort if the found request is in progress. If the specified I/O request is not found or has started processing, an `ATAUnableToAbort` status is returned. If aborted, the `ATAReqAborted` status is returned.

It is up to the application that called the `ATA_Abort` function to clean up the aborted request. Cleaning up includes deallocation of the parameter block and OS reporting.

The manager function code for the `ATA_Abort` function is \$10.

The parameter block associated with this function is defined as follows:

```
struct ATA_Abort                // ATA abort structure
{
    ataPBHdr                    // see definition on page 99
    ATA_PB* AbortPB             // address of the parameter
                                // block to be aborted
    UInt16 Reserved            // reserved
};
typedef struct ATA_Abort ATA_Abort;
```

Field descriptions

<code>ataPBHdr</code>	See the definition of the <code>ataPBHdr</code> parameter block on page 99.
<code>AbortPB</code>	This field contains the address of the I/O parameter block to be aborted.
<code>Reserved</code>	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

Software for ATA Devices

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present
ATAMgrNotInitialized	ATA Manager not initialized
ATAReqAborted	The request was aborted
ATAUnableToAbort	Request to abort couldn't be honored

ATA_BusInquiry

You can use the `ATA_BusInquiry` function to get information about a specific ATA bus. This function is provided for possible future expansion of the Macintosh ATA architecture.

The manager function code for the `ATA_BusInquiry` function is \$03.

The parameter block associated with this function is defined below:

```
struct ATA_BusInquiry           // ATA bus inquiry structure
{
    ataPBHdr                    // see definition on page 99
    UInt16  ataEngineCount;     // ← TBD; 0 for now
    UInt16  ataReserved;       // reserved
    UInt32  ataDataTypes;      // ← TBD; 0 for now
    UInt16  ataIOpbSize;       // ← size of ATA I/O PB
    UInt16  ataMaxIOpbSize;    // ← TBD; 0 for now
    UInt32  ataFeatureFlags;   // ← TBD
    UInt8   ataVersionNum;     // ← HBA Version number
    UInt8   ataHBAINquiry;     // ← TBD; 0 for now
    UInt16  ataReserved2;      // reserved
    UInt32  ataHBAPrivPtr;     // ← pointer to HBA private data
    UInt32  ataHBAPrivSize;    // ← size of HBA private data
    UInt32  ataAsyncFlags;     // ← capability for callback
    UInt32  ataReserved3[4];   // reserved
    UInt32  ataReserved4;     // reserved
    SInt8   ataReserved5[16];  // TBD
    SInt8   ataHBAVendor[16];  // ← HBA Vendor ID
    SInt8   ataContrlFamily[16]; // ← family of ATA controller
    SInt8   ataContrlType[16]; // ← controller model number
    SInt8   ataXPTversion[4];  // ← version number of XPT
    SInt8   ataReserved6[4];   // reserved
    SInt8   ataHBAversion[4];  // ← version number of HBA
    UInt8   ataHBAslotType;    // ← type of slot
    UInt8   ataHBAslotNum;     // ← slot number of the HBA
    UInt16  ataReserved7;      // reserved
    UInt32  ataReserved8;     // reserved
};
typedef struct ATA_BusInquiry ATA_BusInquiry;
```

Software for ATA Devices

Field descriptions

ataPBHdr	See the definition of ataPBHdr on page 99.
ataEngineCount	Currently set to 0.
ataReserved	Reserved. All reserved fields are set to 0.
ataDataTypes	Returns a bitmap of data types supported by this host bus adapter (HBA). The data types are numbered from 0 to 30; 0 through 15 are reserved for Apple definition, and 16 through 30 are available for vendor use. This field is currently not supported; it returns a value of 0.
ataIOpbSize	This field contains the size of the I/O parameter block supported.
ataMaxIOpbSize	This field specifies the maximum I/O size for the HBA. This field is currently not supported and returns 0.
ataFeatureFlags	This field specifies supported features. This field is not supported; it returns a value of 0.
ataVersionNum	The version number of the HBA is returned. The current version returns a value of 1.
ataHBAINquiry	Reserved.
ataHBAPrivPtr	This field contains a pointer to the HBA's private data area. This field is not currently supported; it contains a value of 0.
ataHBAPrivSize	This field contains the byte size of the HBA's private data area. This field is currently not supported; it contains a value of 0.
ataAsyncFlags	These flags indicate which types of asynchronous events the HBA is capable of generating. This field is currently not supported; it contains a value of 0.
ataHBAVendor	This field contains the vendor ID of the HBA. This is an ASCII text field.
ataContrlFamily	Reserved.
ataContrlType	This field identifies the specific type of ATA controller.
ataXPTversion	Reserved.
ataHBAversion	This field specifies the version of the HBA. This field is currently not supported; it contains a value of 0.
ataHBAslotType	This field specifies the type of slot. This field is currently not supported; it contains a value of 0.
ataHBAslotNum	This field specifies the slot number of the HBA. This field is currently not supported; it contains a value of 0.

RESULT CODES

noErr	Successful completion; no error occurred
ATAMgrNotInitialized	ATA Manager not initialized

ATA_DrvrDeregister

You can use the `ATA_DrvrDeregister` function to deregister the selected drive. After successful completion of this function, the driver reference number for the drive is set to 0, indicating that no driver is in control of this device.

This function should be called when the controlling device is no longer available to the registered driver (device ejection) or the device driver is being closed down. Typically, this call is embedded in the `Close()` function of the driver.

The manager function code for the `ATA_DrvrDeregister` function is \$87.

There are two versions of the data structure for registration. The version is identified by the `ataPBVers` field in the parameter block.

Two versions of the parameter block associated with this function are defined below:

```
// version 1 (ataPBVers = 1)
struct ataDrvRegister          // parameter block structure
                               // for ataPBVers = 1
{
    ataPBHdr                    // header information
    SInt16  drvRefNum;          // not used
    UInt16  FlagReserved;      // reserved
    UInt16  deviceNextID;      // not used
    SInt16  Reserved[21];      // reserved for future expansion
};
typedef struct ataDrvRegister ataDrvRegister;

// version 2(ataPBVers = 2)
struct ataDrvRegister          // parameter block structure
                               // for ataPBVers = 2
{
    ataPBHdr                    // header information
    SInt16  drvRefNum;          // → driver reference number
    UInt16  drvFlags;           // → driver flags; set to 0
    UInt16  deviceNextID;      // not used
    SInt16  Reserved;           // reserved -> should be 0
    ProcPtr ataEHandlerPtr      // → event handler routine ptr
    SInt32  drvContext;         // → value to pass in along
                               // with the event handler
    UInt32  ataEventMask;      // → masks of various events
                               // for event handler
    SInt16  Reserved[14];      // reserved for future expansion
};
typedef struct ataDrvRegister ataDrvRegister;
```

Software for ATA Devices

In deregistration of a notify-all driver, the `ataEHandlerPtr` field is used to match the entry (because the `deviceID` field is invalid for registration and deregistration of the notify-all driver). If the driver is registered as both notify-all and for a specific device, the driver must deregister for each separately.

IMPORTANT

Notify-all device drivers must deregister using parameter version 2. ▲

Field descriptions

<code>ataPBHdr</code>	See the <code>ataPBHdr</code> parameter block definition on page 99.
<code>drvRefNum</code>	This field is not used with the deregister function.
<code>drvFlags</code>	No bit definition has been defined for the field. This field shall be set to 0 to ensure compatibility in the future.
<code>deviceNextID</code>	Not used for this function.
Reserved	Reserved. Should be set to 0
<code>ataEHandlerPtr</code>	A pointer to driver's event handler routine. This field is only used for notify-all driver deregistration. This field is not used for all other deregistration. Because this field is used to identify the correct notify-all driver entry, this field must be valid for notify-all driver deregistration.
<code>drvContext</code>	Not used for this function.
<code>ataEventMask</code>	Not used for this function.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>nsDrvErr</code>	Specified device is not present

ATA_DrvrRegister

You can use the `ATA_DrvrRegister` function to register the driver and an event handler for the drive whose reference number is passed in. Any active driver that controls one or more devices through the ATA Manager must register with the manager to insure proper operation and notification of events. The `ATA_DrvrRegister` function should be called only at noninterrupt time.

The first driver to register for the device gets it. All subsequent registrations for the device are rejected. The registration mechanism is used for manager to notify the appropriate driver when events occur. Refer to Table 7-5 on page 112 for possible events.

The manager function code for the `ATA_DrvrRegister` function is \$85.

There are two versions of the data structure for registration. The version is identified by the `ataPBVers` field in the parameter block.

Version two allows a driver to register as a notify-all driver. Registration of a notify-all driver is signaled by a value of -1 in the `deviceID` field of the header and bit 0 of

Software for ATA Devices

drvFlags set to 0. Notify-all driver registration is used if notification of all device insertions is desired. Registered default drivers will be called if no media driver is found on the media. Typically, an INIT driver registers as a notify-all driver. The single driver may register as a notify-all driver, then later register for one or more devices on the bus.

Note

To ensure proper operation, all PCMCIA/ATA and notify-all device drivers must register using version two, which provides event-handling capability. ♦

Two versions of the parameter block associated with this function are defined below:

```
// version 1 (ataPBVers = 1)
struct      ataDrvRegister // parameter block structure
                                     // for ataPBVers = 1
{
    ataPBHdr // header information
    SInt16   drvRefNum; // → driver reference number
    UInt16   FlagReserved; // reserved -> should be 0
    UInt16   deviceNextID; // not used
    SInt16   Reserved[21]; // reserved for future expansion
};
typedef struct ataDrvRegister ataDrvRegister;

// version 2(ataPBVers = 2)
struct      ataDrvRegister // parameter block structure
                                     // for ataPBVers = 2
{
    ataPBHdr // header information
    SInt16   drvRefNum; // → driver reference number
    UInt16   drvFlags; // → driver flags; set to 0
    UInt16   deviceNextID; // not used
    SInt16   Reserved; // reserved; set to 0
    ProcPtr  ataEHandlerPtr // → event handler routine pointer
    SInt32   drvContext; // → value to pass in along with
                                     // the event handler
    UInt32   ataEventMask; // → masks of various events for
                                     // the event handler
    SInt16   Reserved[14]; // reserved for future expansion
};
typedef struct ataDrvRegister ataDrvRegister;
```

Software for ATA Devices

Field descriptions

<code>ataPBHdr</code>	See the <code>ataPBHdr</code> parameter block definition on page 99.
<code>drvRefNum</code>	This field specifies the driver reference number to be registered. This value must be less than 0 to be valid. This field is a don't-care field for registration of a notify-all driver.
<code>FlagReserved</code>	Reserved.
<code>deviceNextID</code>	Not used by this function.
<code>Reserved[21]</code>	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.
<code>ataEHandlerPtr</code>	A pointer to driver's event handler routine. This routine will be called whenever an event happens, and the mask bit for the particular event is set in the <code>ataEventMask</code> field. The calling convention for the event handler is

```
pascal SInt16 (ataEHandlerPtr) (ATAEventRec*);
```

where `ATAEventRec` is defined as follows:

```
typedef struct
{
    UInt16 eventCode;    // → ATA event code
    UInt16 phyDrvRef;   // → ID associated with
                       // the event
    SInt32 drvContext;  // → context passed in
                       // by the driver
} ATAEventRec;
```

See "Notification of Device Events" beginning on page 132 for a list of the ATA event codes.

<code>drvContext</code>	A value to be passed in when the event handler is called. This value will be loaded into <code>ATAEventRec</code> before calling the event handler.
<code>ataEventMask</code>	The mask defined in this field is used to indicate whether the event handler should be called or not, based on the event. The event handler will be called only if the mask for the event has been set. If the mask is not set for an event, the ATA Manager will take no action. Table 7-5 lists the masks have been defined.

Table 7-5 Event masks

Bits	Event mask
\$00	Null event
\$01	Online event: a device has come online
\$02	Offline event: a device has gone offline
\$03	Device removed event: a device has been removed (taken out)
\$04	Reset event: a device has been reset
\$05	Offline request event: a request to take the drive offline
\$06	Eject request event: a request to eject the drive
\$07	Configuration update event: the system configuration has changed
\$08-\$1F	Reserved for future expansion

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present
paramErr	Parameter error detected

ATA_EjectDrive

You can use the `ATA_EjectDrive` function to eject a device from a selected socket. You must make sure that all partitions associated with the device have been dismounted from the desktop.

The manager function code for the `ATA_EjectDrive` function is \$89.

The data structure of the function is as follows:

```
struct ataEject           // configuration parameter block
{
    ataPBHdr              // header information
    UInt16  Reserved[24]; // reserved
};
typedef struct ataEject ataEject;
```

Field descriptions

ataPBHdr	See the <code>ataPBHdr</code> parameter block definition on page 99.
Reserved[24]	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present

ATA_ExecIO

You can use the `ATA_ExecIO` function to perform data I/O transfers to or from an ATA device. Your application must provide all the parameters needed to complete the transaction prior to calling the ATA Manager. Upon return, the parameter block contains the result of the request.

The manager function code for the `ATA_ExecIO` function is \$01.

The parameter block associated with the `ATA_ExecIO` function is defined below:

```
struct ATA_ExecIO          // ATA_ExecIO structure
{
    ataPBHdr               // see definition on page 99
    SInt8   ataStatusReg;  // ← last device status register image
    SInt8   ataErrorReg;   // ← last device error register
                                // (valid if bit 0 of status field set)
    SInt16  ataReserved;   // reserved
    UInt32  BlindTxSize;   // → data transfer size
    UInt8*  ioBuffer;      // ↔ data buffer ptr
    UInt32  ataActualTxCnt; // ← actual number of bytes
                                // transferred
    UInt32  ataReserved2;  // reserved
    devicePB RegBlock;     // → device register images
    UInt8*  packetCDBPtr;  // ATAPI packet command block pointer
    UInt16  ataReserved3[6]; // Reserved
};
typedef struct ATA_ExecIO ATA_ExecIO;
```

Field descriptions

<code>ataPBHdr</code>	See the parameter block definition on page 99.
<code>ataStatusReg</code>	This field contains the last device status register image. See the ATA/IDE specification for status register bit definitions.
<code>ataErrorReg</code>	This field contains the last device error register image. This field is valid only if the error bit (bit 0) of the status register is set. See the ATA/IDE specification for error register bit definitions.
<code>ataReserved</code>	Reserved. All reserved fields are set to 0 for future compatibility.
<code>BlindTxSize</code>	This field specifies the maximum number of bytes that can be transferred for each interrupt or detection of a data request. Bytes are transferred in blind mode (no byte-level handshake). Once an interrupt or a data request condition is detected, the ATA Manager transfers up to the number of bytes specified in the field from or to the selected device. The typical number is 512 bytes.
<code>ioBuffer</code>	This field contains the host buffer address for the number of bytes specified in the <code>ioReqCount</code> field. Upon returning, this field is updated to reflect data transfers. When the <code>SGType</code> bits of the

Software for ATA Devices

ataFlags field are set, this field points to a scatter gather list. The scatter gather list consists of series of IOBlk entries defined as follows:

```
struct IOBlk
{
    UInt8*ioBuffer;        // ↔ data buffer ptr
    UInt32ioReqCount;     // ↔ transfer length
};
typedef struct IOBlk IOBlk;
```

- ioReqCount This field contains the number of bytes to transfer either from or to the buffer specified in ioBuffer. Upon returning, the ioReqCount field is updated to reflect data transfers (0 if successful; otherwise, the number of bytes that remained to be transferred prior to the error condition). When the SGType bits of the ataFlags field are set, the ioReqCount field contains the number of scatter gather entries in the list pointed to by the ioBuffer field.
- ataActualTxCnt This field contains the total number of bytes transferred for this request.
- ataReserved2 This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.
- RegBlock This field contains the ATA device register image structure. Values contained in this structure are written out to the device during the command delivery state. The caller must provide the image before calling the ATA Manager. The ATA device register image structure is defined as follows:

```
struct Device_PB        // device register images
{
    UInt8  Features;    // → features register image
    UInt8  Count;      // ↔ sector count
    UInt8  Sector;     // ↔ sector start/finish
    UInt8  Reserved;   // reserved
    UInt16 Cylinder;  // ↔ cylinder 68000 format
    UInt8  SDH;        // ↔ SDH register image
    UInt8  Command;    // → Command register image
};
typedef struct Device_PB Device_PB;
```

For ATAPI commands, the cylinder image must contain the preferred PIO DRQ packet size to be written out to the cylinder high/low registers during the command phase.

Software for ATA Devices

packetCDBPtr This field contains the packet pointer for ATAPI. The ATAPI bit of the protocol type field must be set for this field to be valid. Setting the ATAPI protocol bit also signals the manager to initiate the transaction without the DRDY bit set in the status register of the device. For ATA commands, this field should contain 0 to ensure future compatibility. The packet structure for the ATAPI command is defined as follows:

```
struct ATAPICmdPacket// ATAPI command packet structure
{
    SInt16 packetSize;// size of command packet
                        // in bytes (exclude size)
    SInt16 command[8]; // the ATAPI command packet
};
typedef struct ATAPICmdPacket ATAPICmdPacket;
```

ataReserved3[6] These fields are reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified logical drive number does not exist
AT_AbortErr	Command aborted bit set in error register
AT_RecalErr	Track 0 not found bit set in error register
AT_WrFltErr	Write fault bit set in status register
AT_SeekErr	Seek complete bit not set upon completion
AT_UncDataErr	Uncorrected data bit set in error register
AT_CorDataErr	Data corrected bit set in status register
AT_BadBlkErr	Bad block bit set in error register
AT_DMarkErr	Data mark not found bit set in error register
AT_IDNFErr	ID-not-found bit set in error register
ATABusy	Selected device busy (BUSY bit set)
ATAMgrNotInitialized	ATA Manager not initialized
ATAPBInvalid	Invalid device base address detected (= 0)
ATAQLocked	I/O queue locked—cannot proceed
ATAREqInProg	I/O channel in use—cannot proceed
ATATransTimeOut	Timeout: transaction time-out detected
ATAUnknownState	Device in unknown state

ATA_FindRefNum

You can use the `ATA_FindRefNum` function to determine whether a driver has been installed for a given device. You pass in a device ID, and the function returns the current driver reference number registered for the given device. A value of 0 indicates that no driver has been registered. The `deviceNextID` field contains a device ID of the next device in the list. The end of the list is indicated with a value of \$FF.

To create a list of all drivers for the attached devices, pass in \$FF for `deviceID`. This causes `deviceNextID` to be filled with the first device in the list. Each successive driver can be found by moving the value returned in `deviceNextID` into `deviceID` until the function returns \$FF in `deviceNextID`, which indicates the end of the list.

The manager function code for the `ATA_FindRefNum` function is \$86.

Two versions of the parameter block associated with this function are defined below:

```
// version 1 (ataPBVers = 1)
struct ataDrvrRegister // parameter block structure
                       // for ataPBVers = 1
{
    ataPBHdr           // header information
    SInt16   drvRefNum; // ← driver reference number
    UInt16   FlagReserved; // reserved; set to 0
    UInt16   deviceNextID; // ← used to specify the
                       // next drive ID
    SInt16   Reserved[21]; // reserved for future expansion
};
typedef struct ataDrvrRegister ataDrvrRegister;

// version 2(ataPBVers = 2)
struct ataDrvrRegister // parameter block structure
                       // for ataPBVers = 2
{
    ataPBHdr           // header information
    SInt16   drvRefNum; // ← driver reference number
    UInt16   drvFlags; // → reserved; set to 0
    UInt16   deviceNextID; // ← used to specify the
                       // next drive ID
    SInt16   Reserved; // reserved -> should be 0
    ProcPtr  ataEHandlerPtr // ← event handler routine pointer
    SInt32   drvContext; // ← value to pass in along with
                       // the event handler
    UInt32   ataEventMask; // ← current setting of the mask
                       // of events for the event handler
    SInt16   Reserved[14]; // reserved for future expansion
};
typedef struct ataDrvrRegister ataDrvrRegister;
```

Software for ATA Devices

Field descriptions

ataPBHdr	See the ataPBHdr parameter block definition on page 99.
drvRefNum	Upon return, this field contains the reference number for the device specified in the deviceID field of the ataPBHdr data.
FlagReserved	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.
deviceNextID	Upon return, this field contains the deviceID value of the next device on the list.
Reserved[21]	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present

ATA_GetDevConfig

You can use the ATA_GetDevConfig function to get the current configuration of a device. The configuration includes current voltage settings and access characteristics. This function can be issued to any bus that the ATA Manager supports. However, some fields returned may not be valid for the particular device type (for example, the voltage settings for the internal device are invalid).

The manager function code for the ATA_GetDevConfig function is \$8A.

The data structure for the function is as follows:

```
struct    ataGetDevConfig    // parameter block
{
    ataPBHdr                // header information
    SInt32    ConfigSetting;    // ↔ socket configuration setting
    UInt8    ataIOSpeedMode;    // reserved for future expansion
    UInt8    Reserved3;        // reserved for word alignment
    UInt16    pcValid;        // ↔ mask indicating which
                                // PCMCIA-unique fields
                                // are valid, when set
    UInt16    RWMultipleCount;    // reserved for future expansion
    UInt16    SectorsPerCylinder; // reserved for future expansion
    UInt16    Heads;            // reserved for future expansion
    UInt16    SectorsPerTrack;    // reserved for future expansion
    UInt16    socketNum;        // ← socket number used by
                                // Card Services
    UInt8    socketType;        // ← specifies the socket type
}
```

Software for ATA Devices

```

    UInt8    deviceType;        // ← specifies the active
                                // device type
    // fields below are valid according to the bit mask
    // in pcValid (PCMCIA unique fields)
    UInt8    pcAccessMode;     // ↔ access mode of the socket:
                                // memory or I/O
    UInt8    pcVcc;           // ↔ Vcc voltage in tenths
    UInt8    pcVpp1;         // ↔ Vpp 1 voltage in tenths
    UInt8    pcVpp2;         // ↔ Vpp 2 voltage in tenths
    UInt8    pcStatus;       // ↔ card status register setting
    UInt8    pcPin;          // ↔ card pin register setting
    UInt8    pcCopy;         // ↔ card socket/copy register
                                // setting
    UInt8    pcConfigIndex;   // ↔ card option register setting
    UInt16   Reserved[10];    // reserved
};
typedef struct ataGetDevConfiguration ataGetDevConfiguration;

```

Field descriptions

ataPBHdr See the `ataPBHdr` parameter block definition on page 99.

ConfigSetting This field indicates various configuration settings. The following bits have been defined:

- Bits 5–0: Reserved for future expansion (set to 0)
- Bit 6: ATAPI packet DRQ handling setting (only applies to ATAPI)
 - 1: The function waits for an interrupt to happen before sending the ATAPI command packet.
 - 0: The function waits for the assertion of DRQ bit in the status register before sending the ATAPI command packet. This is the default setting.
- Bits 7–31: Reserved (set to 0)

ataIOSpeedMode This field is reserved for future expansion.

pcValid This field indicates which of the PCMCIA unique fields contain valid values. Table 7-6 on page 119 lists the fields corresponding to each bit.

RWMultipleCount This field is reserved for future expansion.

SectorsPerCylinder This field is reserved for future expansion.

Heads This field is reserved for future expansion.

SectorsPerTrack This field is reserved for future expansion.

socketNum This field contains the socket number used by Card Services for the device. This value will be needed to request services directly from Card Services (such as `GetTuple`). A value of `$FF` indicates that the selected device is not a Card Services client.

Software for ATA Devices

<code>socketType</code>	This field specifies the type of the socket. Possible values are \$00 = unknown socket type \$01 = internal ATA bus \$02 = media bay socket \$03 = PCMCIA socket
<code>deviceType</code>	This field specifies the type of the device. Possible values are \$00 = unknown type or no device present \$01 = standard ATA device \$02 = ATAPI device \$03 = PCMCIA ATA device
<code>pcAccessMode</code>	This field specifies the current access mode of the device; it is valid only if bit 0 of the <code>pcValid</code> field is set, and only for <code>ATA_GetDeviceConfiguration</code> , not for <code>ATA_SetDeviceConfiguration</code> . Possible values are: 0 = I/O mode 1 = memory mode
<code>pcVcc</code>	This field indicates the current voltage setting of Vcc in tenths of a volt. It is valid only if bit 1 of the <code>pcValid</code> field is set.
<code>pcVpp1</code>	This field indicates the current voltage setting of Vpp1 in tenths of a volt. It is valid only if bit 2 of the <code>pcValid</code> field is set.
<code>pcVpp2</code>	This field indicates the current voltage setting of Vpp2 in tenths of a volt. It is valid only if bit 3 of the <code>pcValid</code> field is set.
<code>pcStatus</code>	This field indicates the current card register setting of the PCMCIA device. It is valid only if bit 4 of the <code>pcValid</code> field is set.
<code>pcPin</code>	This field indicates the current card pin register setting of the PCMCIA device. It is valid only if bit 5 of the <code>pcValid</code> field is set.
<code>pcCopy</code>	This field indicates the current card socket/copy register setting of the PCMCIA device. It is valid only if bit 6 of the <code>pcValid</code> field is set.
<code>pcConfigIndex</code>	This field indicates the current card option register setting of the PCMCIA device. It is valid only if bit 7 of the <code>pcValid</code> field is set.

Table 7-6 Bits in `pcValid` field

Bits	Field validity indicated
0	<code>pcAccessMode</code> field is valid, when set
1	<code>pcVcc</code> field is valid, when set
2	<code>pcVpp1</code> field is valid, when set
3	<code>pcVpp2</code> field is valid, when set
4	<code>pcStatus</code> field is valid, when set
5	<code>pcPin</code> field is valid, when set

continued

Table 7-6 Bits in `pcValid` field (continued)

Bits	Field validity indicated
6	<code>pcCopy</code> field is valid, when set
7	<code>pcConfigIndex</code> field is valid, when set
8–14	Reserved (set to 0)
15	Reserved

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>nsDrvErr</code>	Specified device is not present

ATA_GetDevLocationIcon

You can use the `ATA_GetDevLocationIcon` function to get the location icon data and the icon string for the selected device. The length of the icon data returned is fixed at 256 bytes; the string is delimited by the null character. Both the icon data and location string are copied to buffers pointed to by the structure. Data is not copied if the corresponding pointer is set to 0.

The `locationString` field is in C string format. You may have to call `c2pstr()` function to convert to a Pascal string before returning the string to the operating system.

The manager function code for the `ATA_GetDevLocationIcon` function is `$8C`.

The data structure for the `DrvLocationIcon` function is as follows:

```
struct DrvLocationIcon
{
    ataPBHdr                // see above definition
    SInt16  ataIconType;    // → icon type specifier
    SInt16  ataIconReserved; // reserved; set to 0
    SInt8   *ataLocationIconPtr;
                                // → pointer to icon data buffer
    SInt8   *ataLocationStringPtr;
                                // → pointer to location string
                                // data buffer
    SInt16  Reserved[18];    // reserved
};
typedef struct DrvLocationIcon DrvLocationIcon;
```

Software for ATA Devices

Field descriptions

<code>ataPBHdr</code>	See the <code>ataPBHdr</code> parameter block definition on page 99.
<code>ataIconType</code>	This field defines the type of icon desired as follows: \$01 = large black-and-white icon with mask \$81 = same as 1, but ProDOS icon
<code>ataIconReserved</code>	Reserved to be longword aligned. This field should be set to 0 for future compatibility.
<code>ataLocationIconPtr</code>	A pointer to the location icon buffer. When the pointer is nonzero, the function copies the icon data to the buffer.
<code>ataLocationStringPtr</code>	A pointer to the location string buffer. When the pointer is nonzero, the function copies the string data to the buffer.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>ATAInternalErr</code>	The icon data and string could not be found

ATA_Identify

You can use the `ATA_Identify` function to obtain device identification data from the selected device. The identification data contains information necessary to perform I/O to the device. Refer to the ATA/IDE specification for the format and the information description provided by the data.

The manager function code for the `ATA_Identify` function is \$13.

If the `ATAPI` bit is set in the protocol type field of the header, the ATA Manager performs the ATAPI identify command (\$A1).

The parameter block associated with this function is defined below:

```
struct    ataIdentify    // parameter block structure
{
    ataPBHdr                // see definition on page 99
    SInt8    ataStatusReg; // ← last ATA status image
    SInt8    ataErrorReg;  // ← last ATA error image;
                                // valid if error bit set
    SInt16   ataReserved;  // reserved
    UInt32   BlindTxSize;  // ← this field is set to 512
                                // upon returning
    UInt8    *DataBuf;     // buffer for the identify data
                                // (512 bytes)
```

Software for ATA Devices

```

    UInt32    ataRequestCount; // ← indicates remaining
                                   // byte count
    UInt32    ataActualTxCnt;  // ← actual transfer count
    UInt32    ataReserved2;   // reserved
    devicePB  RegBlock;       // ← task file image sent for
                                   // the command
    UInt16    Reserved3[8];   // used internally by ATA Manager
};
typedef struct ataIdentify ataIdentify;

```

Field descriptions

ataPBHdr See the definition of the `ataPBHdr` parameter block on page 99.

ataStatusReg Status register image for the last ATA task file.

ataErrorReg Error register image for the last ATA task file. This field is only valid if the LSB (error bit) of the `ataStatusReg` field is set.

BlindTxSize Byte size of the identifying data.

DataBuf Pointer to the data buffer for the device identifying data. The length of the buffer must be at least 512 bytes.

ataReserved, ataReserved2, Reserved3[8] These fields are reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>nsDrvErr</code>	Specified device is not present

ATA_MgrInquiry

You can use the `ATA_MgrInquiry` function to get information, such as the version number, about the ATA Manager. This function may be called before initialization of the manager; however, the system configuration information may be invalid.

The manager function code for the `ATA_MgrInquiry` function is \$90.

The parameter block associated with this function is defined below:

```

struct ATA_MgrInquiry // ATA inquiry structure
{
    ataPBHdr // see definition on page 99
    NumVersion MgrVersion // ← manager version number
    UInt8 MGRPBVers; // ← manager PB version number
                                   // supported
    UInt8 Reserved1; // reserved
    UInt16 ataBusCnt; // ← number of ATA buses in system

```

Software for ATA Devices

```

    UInt16    ataDevCnt;        // ← number of ATA devices detected
    UInt8     ataMaxMode;      // ← maximum I/O speed mode
    UInt8     Reserved2;      // reserved
    UInt16    IOClkResolution; // ← I/O clock resolution in ns
    UInt16    Reserved[17];   // reserved
};
typedef struct ATA_MgrInquiry ATA_MgrInquiry;

```

Field descriptions

ataPBHdr	See the ataPBHdr parameter block definition on page 99.
MgrVersion	Upon return, this field contains the version number of the ATA Manager.
MGRPBVers	This field contains the number corresponding to the latest version of the parameter block supported. A client may use any parameter block definition up to this version.
Reserved1	Reserved. All reserved fields are set to 0 for future compatibility.
ataBusCnt	Upon return, this field contains the total number of ATA buses in the system. This field contains 0 if the ATA Manager has not been initialized.
ataDevCnt	Upon return, this field contains the total number of ATA devices detected on all ATA buses. The current architecture allows only one device per bus. This field will contain 0 if the ATA Manager has not been initialized.
ataMaxMode	This field specifies the maximum I/O speed mode that the ATA Manager supports. Refer to the ATA/IDE specification for information on mode timing.
IOClkResolution	This field contains the I/O clock resolution in nanoseconds. The current implementation doesn't support the field (returns 0).
Reserved[17]	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

noErr	0	Successful completion; no error occurred
-------	---	--

ATA_ModifyDrvEventMask

You can use the `ATA_ModifyDrvEventMask` function for modifying an existing driver event mask that has been specified by the `ATA_DrvrRegister` function. Modifying the mask for a nonregistered bus has no effect.

This function is only available with `ataPBVers` of two (2).

The manager function code for the `ATA_ModifyDrvEventMask` function is \$88.

Software for ATA Devices

The data structure of the function is as follows:

```
struct ataModifyEventMask
{
    ataPBHdr                // header information
    UInt32  modifiedEventMask; // → new event mask value
    SInt16  Reserved[22];    // reserved for future expansion
};
typedef struct ataModifyEventMask ataModifyEventMask;
```

Field descriptions

ataPBHdr	See the ataPBHdr parameter block definition on page 99.
modifiedEventMask	New event mask setting. The definitions of the subfields are given in Table 7-5 on page 112.
Reserved[22]	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

noErr	Successful completion; no error occurred
ATAInternalErr	The icon data and string could not be found

ATA_NOP

The ATA_NOP function performs no operation across the interface and does not change the state of either the manager or the device. It returns noErr if the drive number is valid.

The manager function code for the ATA_NOP function is \$00.

The parameter block associated with this function is defined below:

```
lstruct    ataNOP                // parameter block structure
{
    ataPBHdr                // see definition on page 99
    UInt16  Reserved[24];    // reserved
};
typedef struct ataNOP ataNOP;
```

Field descriptions

ataPBHdr	See the definition of ataPBHdr on page 99.
----------	--

There are no additional function-specific variations on ataPBHdr for this function.

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present

ATA_QRelease

You can use the `ATA_QRelease` function to release a frozen I/O queue.

When the ATA Manager detects an I/O error and the `QLockOnError` bit of the parameter block is set for the request, the ATA Manager freezes the queue for the selected device. No pending or new requests are processed or receive status until the queue is released through the `ATA_QRelease` command. Only those requests with the `Immediate` bit set in the `ATAFlags` field of the `ataPBHdr` parameter block are processed. Consequently, for the ATA I/O queue release command to be processed, it must be issued with the `Immediate` bit set in the parameter block. An ATA I/O queue release command issued while the queue isn't frozen returns the `noErr` status.

The manager function code for the `ATA_QRelease` function is \$04.

The parameter block associated with this function is defined as follows:

```
struct ataQRelease           // parameter block structure
{
    ataPBHdr                 // see definition on page 99
    UInt16   Reserved[24];   // reserved
};
typedef struct ataQRelease ataQRelease;
```

Field descriptions

`ataPBHdr` See the definition of `ataPBHdr` on page 99.

There are no additional function-specific variations on `ataPBHdr` for this function.

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present
ATAMgrNotInitialized	ATA Manager not initialized

ATA_RegAccess

You can use the `ATA_RegAccess` function to gain access to a particular device register of a selected device. This function is used for diagnostic and error recovery processes.

The manager function code for the `ATA_RegAccess` function is \$12.

Software for ATA Devices

Two versions of the parameter block associated with this function are defined below:

```
// version 1 (ataPBVers = 1)
struct      ataRegAccess      // parameter block structure
                                     // for ataPBVers of 1
{
    ataPBHdr      // see definition on page 99
    UInt16  RegSelect;      // → device register selector
    union      {
        UInt8  byteRegValue; // ↔ byte register value read
                                     // or to be written
        UInt16 wordRegValue; // ↔ word register value read
                                     // or to be written
    } registerValue;
    UInt16  Reserved[22];    // reserved
};
typedef struct ataRegAccess ataRegAccess;

// version 2 (ataPBVers = 2)
struct      ataRegAccess      // parameter block structure
                                     // for ataPBVers of 2
{
    ataPBHdr      // see definition on page 99
    UInt16  RegSelect;      // → device register selector
    union      {
        UInt8  byteRegValue; // ↔ register value read or
                                     // to be written
        UInt16 wordRegValue; // ↔ word register value read
                                     // or to be written
    } registerValue;
    // The following fields are valid only if RegSelect = $FFFF
    UInt16  regMask;        // → mask indicating which
                                     // combination of registers
                                     // to access.
    devicePB ri;           // ↔ register images
                                     // (feature - command)
    UInt8    altStatDevCntrReg; // ↔ alternate status (R) or
                                     // device control (W) register
    UInt8    Reserved3;     // reserved (set to 0)
    UInt16  Reserved[16];   // reserved
};
typedef struct ataRegAccess ataRegAccess;
```

Field descriptions

<code>ataPBHdr</code>	See the definition of the <code>ataPBHdr</code> parameter block on page 99.
<code>RegSelect</code>	This field specifies which of the device registers to access. The selectors for the registers supported by the <code>ATA_RegAccess</code> function are listed in Table 7-7.
<code>registerValue</code>	This field represents the value to be written (<code>ATAIoDirection = 01</code> binary) or the value read from the selected register (<code>ATAIoDirection = 10</code> binary). For the data register, this field is a 16-bit field; for other registers, an 8-bit field. In the case where the <code>RegSelect</code> field is set to <code>\$FFFF</code> (<code>ataPBVers = 2</code> or higher), this field is used to store the upper byte of the data register image.
<code>Reserved[22]</code>	This field is unused except in the cases where <code>RegSelect</code> is set to either 0 (data register selected) or <code>\$FFFF</code> (more than one register selected). In those two cases, this field contains the lower byte of the data register image.
<code>regMask</code>	This field is only valid for an <code>ataPBVers</code> value of 2 or higher. This field indicates what combination of the taskfile registers should be accessed. A bit set to one indicates either a read or a write to the register. A bit set to zero performs no operation to the register. Bit assignments are as shown in Table 7-8.
<code>ri</code>	This field is only valid for an <code>ataPBVers</code> value of 2 or higher. This field contains register images for error/features, sector count, sector number, cylinder low, cylinder high, SDH, and status/command. Only those register images indicated in the <code>regMask</code> field are valid. See “ATA_ExecIO” on page 113 for the structure definition.
<code>altStatDevCntrReg</code>	This field is only valid for <code>ataPBVers</code> value of 2 or higher. This field contains the register image for alternate status (R) or device control (W) register. This field is valid if the alternate status/device control register bit in the <code>regMask</code> field is set to 1.

Table 7-7 ATA register selectors

Selector name	Selector	Register description
<code>DataReg</code>	0	Data register (16-bit access only)
<code>ErrorReg</code>	1	Error register (R) or features register (W)
<code>SecCntReg</code>	2	Sector count register
<code>SecNumReg</code>	3	Sector number register
<code>CylLoReg</code>	4	Cylinder low register
<code>CylHiReg</code>	5	Cylinder high register
<code>SDHReg</code>	6	SDH register
<code>StatusReg</code> <code>CmdReg</code>	7	Status register (R) or command register (W)

continued

Software for ATA Devices

Table 7-7 ATA register selectors (continued)

Selector name	Selector	Register description
AltStatus DevCntr	\$0E	Alternate status (R) or device control (W)
	\$FFFF	More than one register access (valid only for ataPBVers = 2 or higher)

Table 7-8 Register mask bits

Bit number	Definition
0	Data register
1	Error register (R) or feature register (W)
2	Sector count register
3	Sector number register
4	Cylinder low register
5	Cylinder high register
6	SDH register
7	Status register (R) or command register (W)
8–13	Reserved (set to 0)
14	Alternate status register (R) or device control register (W)
15	Reserved (set to 0)

When reading or writing ATA registers, use the following order:

1. Data register
2. Alternate status register (R) or device control register (W)
3. Error register (R) or feature register (W)
4. Sector count register
5. Sector number register
6. Cylinder low register
7. Cylinder high register
8. Status register (R) or command register (W)

RESULT CODES

noErr	Successful completion; no error occurred
nsDrvErr	Specified device is not present

ATA_ResetBus

You can use the `ATA_ResetBus` function to reset the specified ATA bus. This function performs a soft reset operation to the selected ATA bus. The ATA interface doesn't provide a way to reset individual units on the bus. Consequently, all devices on the bus will be reset.

The manager function code for the `ATA_ResetBus` function is \$11.

IMPORTANT

You should avoid calling this function under interrupt because it may take up to several seconds to complete. ▲

▲ WARNING

Use this function with caution; it may terminate any active requests to devices on the bus. ▲

If the `ATAPI` bit is set in the protocol type field of the header, the ATA Manager will perform the ATAPI reset command (\$08).

Upon completion, this function flushes all I/O requests for the bus in the queue. Pending requests are returned to the client with the `ATAAbortedDueToRst` status.

The parameter block associated with this function is defined below:

```
struct ATA_ResetBus      // ATA reset structure
{
    ataPBHdr              // see definition on page 99
    SInt8   Status;      // ← last ATA status register image
    SInt8   Reserved;    // reserved
    UInt16  Reserved[23]; // reserved
};
typedef struct ATA_ResetBus ATA_ResetBus;
```

Field descriptions

<code>ataPBHdr</code>	See the definition of the <code>ataPBHdr</code> parameter block on page 99.
<code>Status</code>	This field contains the last device status register image following the bus reset. See the ATA/IDE specification for definitions of the status register bits.
<code>Reserved[23]</code>	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>nsDrvErr</code>	Specified device is not present

ATA_SetDevConfig

You can use the `ATA_SetDevConfig` function to set the configuration of a device. It contains the current voltage setting and access characteristics. This function can be issued to any bus that the ATA Manager controls. However, some field settings may be inappropriate for the particular device type (for example, setting the voltage for the internal device).

The manager function code for the `ATA_SetDevConfig` function is \$8B.

The `ataSetDevConfiguration` data structure of the `ATA_SetDevConfig` function is as follows:

```
struct ataSetDevConfiguration // configuration parameter block
{
    ataPBHdr           // header information
    SInt32  ConfigSetting; // ↔ socket configuration setting
    UInt8   ataIOSpeedMode; // reserved for future expansion
    UInt8   Reserved3;     // reserved for word alignment
    UInt16  pcValid;      // ↔ mask indicating which
                                // PCMCIA-unique fields are valid
    UInt16  RWMultipleCount; // reserved for future expansion
    UInt16  SectorsPerCylinder; // reserved for future expansion
    UInt16  Heads;        // reserved for future expansion
    UInt16  SectorsPerTrack; // reserved for future expansion
    UInt16  Reserved4[2]; // reserved
    // fields below are valid according to the bit mask
    // in pcValid (PCMCIA unique fields)
    UInt8   pcAccessMode; // ↔ access mode of the socket:
                                // memory or I/O
    UInt8   pcVcc;        // ↔ Vcc voltage
    UInt8   pcVpp1;      // ↔ Vpp 1 voltage
    UInt8   pcVpp2;      // ↔ Vpp 2 voltage
    UInt8   pcStatus;    // ↔ card status register setting
    UInt8   pcPin;       // ↔ card pin register setting
    UInt8   pcCopy;      // ↔ card socket/copy register
                                // setting
    UInt8   pcConfigIndex; // ↔ card option register setting
    UInt16  Reserved[10]; // reserved
};
typedef struct ataSetDevConfiguration ataSetDevConfiguration;
```

Software for ATA Devices

Field descriptions

<code>ataPBHdr</code>	See the <code>ataPBHdr</code> parameter block definition on page 99.
<code>ConfigSetting</code>	This field controls various configuration settings. The following bits have been defined: Bits 0–5: Reserved for future expansion (set to 0) Bit 6: ATAPI packet DRQ handling setting (applies only to ATAPI) 1 = The function waits for an interrupt to happen before sending the ATAPI command packet. 0 = The function waits for the assertion of the DRQ bit in the status register before sending the ATAPI command packet. This is the default setting. Bits 7–31: Reserved (set to 0)
<code>ataIOSpeedMode</code>	This field is reserved for future expansion.
<code>pcValid</code>	This field indicates which of the PCMCIA unique fields contain valid values. Table 7-6 on page 119 lists the fields corresponding to each bit.
<code>RWMultipleCount</code>	This field is reserved for future expansion.
<code>SectorsPerCylinder</code>	This field is reserved for future expansion.
<code>Heads</code>	This field is reserved for future expansion.
<code>SectorsPerTrack</code>	This field is reserved for future expansion.
<code>pcAccessMode</code>	This field is valid only if the bit 0 of the <code>pcValid</code> field is set. The value is written to the access mode control. Possible values are 0 = I/O mode 1 = memory mode
<code>pcVcc</code>	This field indicates the new voltage setting of Vcc in tenths of a volt. It is valid only if bit 1 of the <code>pcValid</code> field is set.
<code>pcVpp1</code>	This field indicates the new voltage setting of Vpp1 in tenths of a volt. It is valid only if bit 2 of the <code>pcValid</code> field is set.
<code>pcVpp2</code>	This field indicates the new voltage setting of Vpp2 in tenths of a volt. It is valid only if bit 3 of the <code>pcValid</code> field is set.
<code>pcStatus</code>	This field indicates the new card register setting of the PCMCIA device. It is valid only if bit 4 of the <code>pcValid</code> field is set.
<code>pcPin</code>	This field indicates the new card pin register setting of the PCMCIA device. It is valid only if bit 5 of the <code>pcValid</code> field is set.
<code>pcCopy</code>	This field indicates the new card socket/copy register setting of the PCMCIA device. It is valid only if bit 6 of the <code>pcValid</code> field is set.
<code>pcConfigIndex</code>	This field indicates the new card option register setting of the PCMCIA device. It is valid only if bit 7 of the <code>pcValid</code> field is set.

RESULT CODES

<code>noErr</code>	Successful completion; no error occurred
<code>nsDrvErr</code>	Specified device is not present

Using the ATA Manager With Drivers

This section describes several operations dealing with drivers:

- notification of device events
- loading a device driver
- old and new driver entry points
- loading a driver from the media
- notification of notify-all drivers
- notification of the ROM driver

Notification of Device Events

Due to the asynchronous event-reporting mechanism of the Card Services Manager, the ATA Manager notifies its clients by a callback mechanism using the client's event handler. Each client that is to be notified of device events must register its event handler at the time of driver registration. Refer to the section "ATA_DrvrRegister" beginning on page 109 for the calling convention of the event handler.

The event codes that have been defined are listed in Table 7-9.

Table 7-9 Event codes send by the ATA Manager

Event code	Event description
\$00	Null event; signifies no real event. The client should simply return with no error code.
\$01	<p>Online event; signifies that a device has come online. This event may happen as a result of the following actions:</p> <ul style="list-style-type: none"> ■ A device has been inserted into the socket. ■ A device has been repowered from sleep/low power. <p>The client should let the operating system know about the presence of the device (if it has not done so already), verify the device type, and upload any device characteristics.</p>
\$02	<p>Offline event; signifies that the device has gone offline. This event may happen as a result of a device being manually removed from the socket.</p> <p>The client should let the operating system know that the device has gone offline by setting the offline bit, if appropriate.</p>
\$03	<p>Device-removed event; signifies that the device has been ejected gracefully. The client should clean up the internal variables to reflect the latest state of the socket. The client may notify the operating system of the event.</p>

continued

Table 7-9 Event codes send by the ATA Manager (continued)

Event code	Event description
\$04	Reset event; signifies that the device has been reset. This indicates that any pending request or the settings may have been aborted.
\$05	Offline request event; requests permission for the device to go offline.
\$06	Eject request event; requests permission to eject the drive.
\$07	Configuration update event; signifies that the system configuration related to I/O subsystems may have changed. This event may imply that the number of ATA buses and devices has changed. Consequently, if the client is a driver capable of handling more than one device, it may want to query the manager for the current configuration.

Device Driver Loading

This section describes the sequence and method of driver installation and the recommended driver initialization sequence.

The operating system attempts to install a device driver for a given ATA device in the following instances:

- during system startup or restart
- during accRun, following the drive insertion
- each time it is called to register a notify-all driver

Three classes of drivers are identified and discussed below. The driver loading and initialization sequence is as follows:

1. Media driver. The driver on the media is given the highest priority.
2. notify-all drivers. Any INIT drivers are given the next priority.
3. ROM driver. The built-in ROM driver is loaded if no other driver is found.

The initialization sequences for the three driver classes are described in “Loading a Driver From the Media” on page 135.

Once the driver loading and initialization sequence has been performed for a particular device, the process is not repeated until one of the following situations occurs:

- system restart
- device ejection followed by an insertion
- shutdown and reinitialization of the manager; but only if the `existingGlobalPtr` field of the parameter block is invalid
- a notify-all driver registration occurs. In this case, only the registering driver is notified of the drive online.

New API Entry Point for Device Drivers

Two entry points into each ATA driver are currently defined, for the old API and the new API. Use of the new API is strongly recommended. The differences between the two APIs are as follows:

- **Entry point:** In the old API, the entry point is offset 0 bytes from the start of the driver; in the new API, it is offset 8 bytes from the start of the driver (the same as with SCSI drivers).
- **D5 register:** In the old API, the input parameter in the D5 register contains just the bus ID; in the new API, the D5 register contains the `devIdent` parameters.

Table 7-10 shows the contents of the D5 register, high-order bits first, for the old API (calls offset 0 bytes into the driver).

Table 7-10 Input parameter bits for the old API

Bits	Value	Definition
31–24	0	Reserved; set to 0.
23–16	0	Reserved; set to 0.
15–8	0	Reserved; set to 0.
7–0	ATA bus ID	The bus ID where the device resides. This is the ID used to communicate with the ATA Manager.

Table 7-11 shows the contents of the D5 register, high-order bits first, for the new API (calls offset 8 bytes into the driver).

Table 7-11 Input parameter bits for the new API

Bits	Value	Definition
31–24	Reserved	In this byte, bits 29–31 are currently defined. All other bits should be set to 0. Bit 31 1 = load at run time (RAM based) 0 = load at startup time (ROM based) Bit 30 1 = mount volumes associated with this drive 0 = don't mount any volume associated with this drive Bit 29 1 = new API entry (use 8-byte offset) 0 = old API entry (use 0-byte offset) This bit is set internally by each driver.
23–16	ATA bus ID	The bus ID where the device resides. This is the ID used to communicate with the ATA Manager.
15–8	Device ID	The device ID within the given bus. This field is used to identify the device on a particular bus. The current and previous ATA Manager implementations assume that the device ID field is always 0.
7–0	Reserved	Reserved; set to 0.

IMPORTANT

ATA Manager version 1.0 uses the old API; the ATA Manager version 2.0 uses the new API. ▲

Loading a Driver From the Media

Upon detection of a device insertion, the driver loader, an extension of the ATA Manager, initiates a driver load operation during `accRun` time. The driver loader searches the DDM and partition maps of the media. If an appropriate driver is found, the driver loader allocates memory in the system heap and loads the driver.

For the old API, the driver is opened by jumping to the first byte of the driver code with the D5 register containing the bus ID of the device. For the new API, the driver is opened by jumping to the eighth byte of the driver code with the D5 register containing the new API definition.

The appropriate driver is identified by following fields:

- `ddType = $701` for Mac OS
- `partition name = Apple_Driver_ATA`

The media driver should be capable of handling both old and new APIs. The Macintosh Quadra 630 uses the old API; other Macintosh models use the new API.

The typical sequence of the media driver during the `Open()` call is as follows:

1. Allocate driver globals.
2. Initialize the globals.
3. Install any system tasks, such as VBL, time manager, shutdown procedure, and the like. Initialize the device and its parameters.
4. Register the device with the ATA Manager. The driver is expected to fail the `Open()` operation if an error is returned from the driver registration call for a given device.

The installed driver is expected to return the following information in D0:

- The upper 16-bit word contains the driver reference number corresponding to the unit table entry. This field is only valid when the lower 16-bits of D0 is 0. The reference number returned must be less than 0 to be valid.
- The lower 16-bit word contains the status of the driver `Open()` operation. A value of 0 indicates no error.

Notify-all Driver Notification

When an error is returned from the media driver loading, the driver load function then calls the notify-all drivers, one by one. This driver type is determined from the driver registration (-1 in the `deviceID` field of the driver registration parameter block). Unlike the media driver, this driver is notified of a device insertion by means of the callback mechanism at `accRun` time, when the manager calls the driver with an online event. Consequently, each notify-all driver must provide a callback routine pointer in the driver registration. The driver may get a series of online event notifications during the notify-all registration. The driver is assumed to be installed in system (for example, the INIT

Software for ATA Devices

driver). Refer to “Notification of Device Events” on page 132 for the event opcode and the definition of the structure passed in.

Upon returning from the call, each notify-all driver must provide a status indicating whether the driver controls the specified device or not. A status of 0 indicates that the driver controls the device; a nonzero status indicates that the driver doesn't control the device.

The calling of the notify-all drivers continues until a 0 status is received from one of the registered drivers or until the end of the list is reached.

The typical sequence of the notify-all driver during the online event handling is as follows:

1. Check for the presence and the device type.
2. If the driver controls this device, allocate and initialize global variables.
3. Initialize the device and its parameters.
4. Perform driver registration for the device with the manager. The driver should release its ownership of the device and return a nonzero status if the driver registration fails.

ROM Driver Notification

If no driver indicates that it controls the device, the ATA Manager calls the ATA hard disk driver in the system ROM. The ROM driver is called only for a hard disk device. For the Macintosh Quadra and LC 630 models, as in the case of the media driver, the called address is the first byte of the driver. For all other Macintosh models, the called address is offset by 8 bytes. The input and the output of the driver and the `Open ()` sequence are the same for both the media driver and the ROM driver.

Device Driver Purging

When a device removal event is detected, an attempt is made to close the device, to remove it from the unit table, and to dispose of the corresponding driver in memory. A key function in supporting this feature is a new driver gestalt call. Driver support for this call is strongly recommended.

The driver gestalt selector for the function is 'purg'. The call provides following information to the driver loader:

- the starting location of the driver
- the purge permissions: `close ()`, `DrvRemove ()`, and `DisposePtr ()`

Software for ATA Devices

The following structure describes the response associated with the purge call. The description of this and other driver gestalt calls can be found in the Driver Gestalt documentation in *Designing PCI Cards and Drivers for Power Macintosh computers*.

```
struct DriverGestaltPurgeResponse
    // driver purge permission structure
{
    SInt16    purgePermission; // <--: purge response
                                // 0 = do not change the
                                // state of the driver
                                // 3 = close and remove
                                // this driver refnum,
                                // but don't deallocate
                                // driver code
                                // 7 = close,remove, and
                                // dispose of pointer

    SInt16    purgeReserved;

    UniversalProcPtr purgeDrvPtr; // <--: starting address
                                // of the driver
                                // (valid only if disposePtr
                                // permission is given)
};
```

The driver must either return a status error indicating that the call is not supported, or return one of the three values defined in the `purgePermission` field of the response structure described above. If an error or an illegal value is returned in response to the call, then the manager treats as if the response of 0 is received. The three possible purge permissions are listed in Table 7-12. All other response values are reserved and should not be used.

Table 7-12 Purge permissions and responses

Response	Purge permissions		
	Close()	DrvPtrRemove()	DisposePtr()
7	√	√	√
3	√	√	
0			

Software for ATA Devices

Upon receiving a response, the manager purge sequence is as follows:

```

if a response of 3 or 7
    if ( (err = PBClose() ) == noErr )
        /* close the driver down*/
    {
        if ( a response of 7)
            DisposePtr ();          /* dispose the driver memory*/
            DrvrRemove ();          /* remove the driver from
                                   the unit table*/
    }

```

The driver `Close()` permission applies only to the corresponding unit table entry. In other words, if the driver is used to control multiple devices (such as multiple unit table entries), then `Close()` should apply only to the particular device with the matching driver reference number. The other devices must remain operational.

The registered driver must make the decision as to what value to return in response to the call. Some examples are listed below:

- If the driver is in control of any other device, it should return a response of 3; the driver closes the particular device down, but the driver stays resident for other devices.
- If the driver must remain available for other potential device insertion, it should return a response of 3.
- If the driver is a media driver controlling the particular device, it should return a response of 7. Another media driver will become active when a device is inserted.

Setting the I/O Speed

The ATA controllers used in Macintosh systems have their I/O cycle time adjustable to optimize data transfers. There are two mechanisms for setting the I/O cycle time: the `ataIOSpeed` field of the parameter block header (this field is only valid when a data transfer is involved: and the `ataIOSpeedMode` field of the `ATA_SetDevConfig` function. The speed setting via the `ATA_SetDevConfig` function is considered the default setting. In other words, if the current speed bit of the `ataFlags` field in the parameter block header is set, then the default speed setting previously set through the `ATA_SetDevConfig` function is used as the I/O speed mode of the particular transaction.

If the current speed bit is cleared, then the speed setting specified in the `ataIOSpeed` field of the transaction parameter block is used. The initial speed setting prior to the first call to `ATA_SetDevConfig` is mode 0.

Because the current PC card specification defines the ATA I/O timing of 0 for all PCMCIA/ATA devices, the speed setting field has no effect on the I/O speed for those devices. Currently the field is hard-coded to mode 0.

Error Code Summary

Table 7-13 lists two sets of error codes for ATA drivers: old error codes, used with the Macintosh PowerBook 150 and the Macintosh Quadra and LC 630 series computers; and new error codes, to be used with all future Macintosh models. The choice of error codes is determined by the `ataPBVers` field in the `ataPBHdr` structure, defined on page 99. If `ataPBVers` is set to 1, then the old error codes are used; if `ataPBVers` is set to 2, then the new error codes are used.

Table 7-13 ATA driver error codes

Error code (new)	Error code (old)	Error name	Error description
0	0	<code>noErr</code>	No error was detected on the requested operation.
\$FFCE (-50)	\$FFCE (-50)	<code>paramErr</code>	Error in parameter block.
\$FFC8 (-56)	\$FFC8 (-56)	<code>nsDrvErr</code>	No such drive; no device is attached to the specified port.
\$DB43 (-9405)	\$F901 (-1791)	<code>AT_NRDyErr</code>	Drive ready condition was not detected.
\$DB44 (-9404)	\$F904 (-1788)	<code>AT_IDNFErr</code>	Sector ID not-found error reported by device.
\$DB45 (-9403)	\$F905 (-1787)	<code>AT_DMarkErr</code>	Data mark not-found error was reported by the device.
\$DB46 (-9402)	\$F906 (-1786)	<code>AT_BadBlkErr</code>	A bad block was detected by the device.
\$DB47 (-9401)	\$F907 (-1785)	<code>AT_CorDataErr</code>	Notification that data was corrected (good data).
\$DB48 (-9400)	\$F906 (-1784)	<code>AT_UncDataErr</code>	Unable to correct data (possibly bad data).
\$DB49 (-9399)	\$F909 (-1783)	<code>AT_SeekErr</code>	A seek error was detected by the device.
\$DB4A (-9398)	\$F90A (-1782)	<code>AT_WrFltErr</code>	A write fault was detected by the device.
\$DB4B (-9397)	\$F90B (-1781)	<code>AT_RecalErr</code>	A recalibration failure was detected by the device.
\$DB4C (-9396)	\$F90C (-1780)	<code>AT_AbortErr</code>	A command was aborted by the device.

continued

Table 7-13 ATA driver error codes (continued)

Error code (new)	Error code (old)	Error name	Error description
\$DB4D (-9395)	\$F90E (-1778)	AT_MCErr	Media-changed error detected by the device.
\$DB4E (-9394)	\$F90F (-1777)	ATAPICheckErr	The ATAPI check condition was detected.
\$DB70 (-9360)	\$F8F6 (-1802)	ATAMgrNotInitialized	ATA Manager has not been initialized. The request function cannot be performed until the manager has been initialized.
\$DB71 (-9359)	\$F8F5 (-1803)	ATAPBInvalid	An invalid ATA port address was detected (ATA Manager initialization problem).
\$DB72 (-9358)	\$F8F4 (-1804)	ATAFuncNotSupported	An unknown ATA Manager function code has been specified.
\$DB73 (-9357)	\$F8F3 (-1805)	ATABusy	The selected device is busy; it is not ready to go to the next phase yet.
\$DB74 (-9356)	\$F8F2 (-1806)	ATATransTimeOut	A timeout condition was detected. The operation had not completed within the user-specified time limit.
\$DB75 (-9355)	\$F8F1 (-1807)	ATAReqInProg	Device busy; the device on the port is busy processing another command.
\$DB76 (-9354)	\$F8F0 (-1808)	ATAUnknownState	The device status register reflects an unknown state.
\$DB77 (-9353)	\$F8EF (-1809)	ATAQLocked	I/O queue for the port is locked due to a previous I/O error. It must be unlocked prior to continuing.
\$DB78 (-9352)	\$F8EE (-1810)	ATAReqAborted	The I/O queue entry was aborted due to an abort command.
\$DB79 (-9351)	\$F8ED (-1811)	ATAUnableToAbort	The I/O queue entry could not be aborted. It was too late to abort or the entry was not found.
\$DB7A (-9350)	\$F8EC (-1812)	ATAAbortedDueToRst	The I/O queue entry aborted due to a bus reset.
\$DB7B (-9349)	\$F8EB (-1813)	ATAPIPhaseErr	Unexpected phase detected.
\$DB7C (-9348)	\$F8EA (-1814)	ATAPIExCntErr	Warning: overrun/underrun condition detected (the data is valid).
\$DB7D (-9347)	\$F8E9 (-1815)	ATANoClientErr	No client present to handle the event.

continued

Table 7-13 ATA driver error codes (continued)

Error code (new)	Error code (old)	Error name	Error description
\$DB7E (-9346)	\$F8E8 (-1816)	ATAInternalErr	Card Services returned an error.
\$DB7F (-9345)	\$F8E7 (-1817)	ATABusErr	A bus error was detected on I/O.
\$DB80 (-9344)	\$F90D (-1818)	AT_NoAddrErr	The task file base address is not valid.
\$DB81 (-9343)	\$F8F9 (-1799)	DriverLocked	The current driver must be removed before adding another.
\$DB82 (-9342)	\$F8F8 (-1800)	CantHandleEvent	Particular event could not be handled.
\$DB83 (-9341)	—	ATAMgrMemoryErr	ATA Manager memory allocation error.
\$DB84 (-9340)	—	ATASDFailErr	ATA Manager shutdown process failed.
\$DB90 (-9328)	—	ATAInvalidDrvNum	Invalid drive number from event.
\$DB91 (-9327)	—	ATAMemoryErr	Memory allocation error.
\$DB92 (-9326)	—	ATANoDDMErr	No driver descriptor map (DDM) was found on the media.
\$DB93 (-9325)	—	ATANoDriverErr	No driver was found on the media.

PC Card Services

PC Card Services

This chapter describes the Card Services part of the PC Card Manager in the Macintosh PowerBook 190 computer.

The PC Card Manager is a new part of Mac OS that lets software use PC cards. The PC Card Manager helps client software recognize, configure, and view PC cards that are inserted into PC card sockets on Macintosh PowerBook computers.

The PC Card Manager comprises two sets of system software:

- Card Services, used by all PC card client software
- Socket Services, used primarily by developers of new PC card hardware

This chapter covers only the Card Services functions. For descriptions of the other functions of the PC Card Manager, see *Developing PC Card Software for the Mac OS*.

Client Information

You can use the functions described in this section to get information about Card Services clients.

The Card Services software keeps information about all its clients in a first-in, first-out queue called the global client queue. You can use the `CSGetFirstClient` and `CSGetNextClient` functions to iterate through all the registered clients. Either of those functions returns a handle that you can then use with the `CSGetClientInfo` function to obtain the corresponding client information.

In the definitions that follow, an arrow preceding a parameter indicates whether the parameter is an input parameter, an output parameter, or both.

Arrow	Meaning
→	Input
←	Output
↔	Both

CSGetFirstClient

You can use the `CSGetFirstClient` function to find the first client in the Card Service's global client queue.

```
pascal OSErr CSGetFirstClient(GetClientPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetClientPB GetClientPB;
struct GetClientPB
{
    UInt32 clientHandle; // ← clientHandle for this client
```

PC Card Services

```

    UInt16  socket;          // → logical socket number
    UInt16  attributes;     // → bitmap of attributes
};

// 'attributes' field values

enum
{
    csClientsForAllSockets = 0x0000,
    csClientsThisSocketOnly = 0x0001
};

```

DESCRIPTION

The `CSGetFirstClient` function returns a `clientHandle` value to the first client in Card Services' global client queue. If the caller specifies `csClientsThisSocketOnly` and passes in a valid socket number, Card Services returns the first client whose event mask for the given socket is not `NULL`.

RESULT CODES

<code>SUCCESS</code>	No error
<code>BAD_SOCKET</code>	Invalid socket specified
<code>NO_MORE_ITEMS</code>	No clients registered

CSGetNextClient

You can use the `CSGetNextClient` function to find the next client in the Card Service's global client queue.

```
pascal OSErr CSGetNextClient(GetClientPB *pb);
```

The parameter block associated with this function is as follows:

```

typedef struct GetClientPB GetClientPB;
struct GetClientPB
{
    UInt32  clientHandle;   // ↔ clientHandle for this client
    UInt16  socket;        // → logical socket number
    UInt16  attributes;    // → bitmap of attributes
};

```

For `attributes` field values, see “`CSGetFirstClient`” on page 144.

PC Card Services

DESCRIPTION

The `CSGetNextClient` function returns the next `clientHandle` value in Card Services' global client queue. If the caller specifies `csClientsThisSocketOnly` and passes in a valid socket number, Card Services returns the next client whose event mask for the given socket is not `NULL`.

RESULT CODES

<code>SUCCESS</code>	No error
<code>BAD_SOCKET</code>	Invalid socket specified
<code>NO_MORE_ITEMS</code>	No clients registered
<code>BAD_HANDLE</code>	Invalid <code>clientHandle</code>

CSGetClientInfo

You can use the `CSGetClientInfo` function to get information from the Card Service's global client queue.

```
pascal OSErr CSGetClientInfo(GetClientInfoPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetClientInfoPB GetClientInfoPB;
struct GetClientInfoPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt16 attributes; // ↔ subfunction + bitmapped client attributes

    union
    {
        struct // upper byte of attributes is
              // csClientInfoSubfunction
        {
            UInt16 revision; // ← BCD value of client's revision
            UInt16 csLevel; // ← BCD value of CS release
            UInt16 revDate; // ← revision date:
                          // y[15-9], m[8-5], d[4-0]
            SInt16 nameLen; // ↔ in: maximum length of
                          // client name string,
                          // out: actual length
            SInt16 vStringLen; // ↔ in: max length of vendor string,
                              // out: actual length
            UInt8 *nameString; // ← pointer to client name string
                              // (zero-terminated)
            UInt8 *vendorString; // ← pointer to vendor string
                              // (zero-terminated)
        }
        ClientInfo;
    }
};
```

PC Card Services

```

    struct                // upper byte of attributes is
                        // csCardNameSubfunction,
    {                    // csCardTypeSubfunction,
                        // csHelpStringSubfunction
        UInt16  socket; // → logical socket number
        UInt16  reserved; // → zero
        SInt16  length; // ↔ in: max length of string,
                        //      out: actual length
        UInt8   *text; // ← pointer to string (zero-terminated)
    }
    AlternateTextString;

    struct                // upper byte of attributes is
                        // csCardIconSubfunction
    {
        UInt16  socket; // → logical socket number
        Handle  iconSuite; // ← handle to suite containing all icons
    }
    AlternateCardIcon;

    struct                // upper byte of attributes is
                        // csActionProcSubfunction
    {
        UInt16  socket; // → logical socket number
    }
    CustomActionProc;

} u;
};

// 'attributes' field values

enum {
    csMemoryClient          = 0x0001,
    csIOClient              = 0x0004,
    csClientTypeMask        = 0x0007,
    csShareableCardInsertEvents = 0x0008,
    csExclusiveCardInsertEvents = 0x0010,

    csInfoSubfunctionMask   = 0xFF00,
    csClientInfoSubfunction = 0x0000,
    csCardNameSubfunction   = 0x8000,
    csCardTypeSubfunction   = 0x8100,
    csHelpStringSubfunction = 0x8200,
    csCardIconSubfunction   = 0x8300,
    csActionProcSubfunction = 0x8400
};

```

PC Card Services

DESCRIPTION

The `CSGetClientInfo` function is used to obtain information about a client from the Card Service's global client queue. The client is specified by passing in a `clientHandle` value previously obtained using `GetFirstClient` or `GetNextClient`.

Note that in this case the caller does not pass in its own `clientHandle` value, but that of the client whose information is being requested.

The caller of the `CSGetClientInfo` function specifies the type of information being requested by setting the requested information subfunction in the upper byte of the `attributes` field. The Card Services software passes a `CLIENT_INFO` message to the client pointed to by `clientHandle`. Called clients are expected to respond to the `CLIENT_INFO` message by providing the data requested. When a client receives a `CLIENT_INFO` message to perform a custom action, it needs to be aware that it is being called from the Finder or a similar process environment.

Each time the Card Services software calls a client with a `CLIENT_INFO` message, Card Services passes a client callback parameter block (`ClientCallbackPB`). The `buffer` field of the `ClientCallbackPB` structure contains a pointer to the get client info parameter block (`GetClientInfoPB`), which has the following structure:

```
ClientCallbackPB.function      = CLIENT_INFO;
ClientCallbackPB.socket       = 0;
ClientCallbackPB.info        = 0;
ClientCallbackPB.misc        = 0;
ClientCallbackPB.buffer      = (Ptr) GetClientInfoPB;

ClientCallbackPB.clientData
= ((ClientQRecPtr) GetClientInfoPB->clientHandle)->clientDataPtr;
```

Before calling the `CSGetClientInfo` function, you should use `GetFirstClient` and `GetNextClient` to iterate through the registered clients. Card Services returns `clientHandle` to the caller of either function.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid <code>clientHandle</code> value

Configuration

The functions described in this section help you configure cards and sockets.

CSGetConfigurationInfo

You can use the CSGetConfigurationInfo function to get the information needed to initialize a CSModifyConfiguration parameter block.

```
pascal OSErr
    CSGetConfigurationInfo(GetModRequestConfigInfoPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  socket;      // → logical socket number
    UInt16  attributes;  // ← bitmap of configuration attributes
    UInt8   vcc;         // ← Vcc setting
    UInt8   vpp1;        // ← Vpp1 setting
    UInt8   vpp2;        // ← Vpp2 setting
    UInt8   intType;     // ← interface type (memory or memory+I/O)
    UInt32  configBase;  // ← card base address of config registers
    UInt8   status;     // ← card status register setting, if present
    UInt8   pin;        // ← card pin register setting, if present
    UInt8   copy;       // ← card socket/copy reg setting, if present
    UInt8   configIndex; // ← card option register setting, if present
    UInt8   present;    // ← bitmap of which config regs are present
    UInt8   firstDevType; // ← from DeviceID tuple
    UInt8   funcCode;   // ← from FuncID tuple
    UInt8   sysInitMask; // ← from FuncID tuple
    UInt16  manufCode;  // ← from ManufacturerID tuple
    UInt16  manufInfo;  // ← from ManufacturerID tuple
    UInt8   cardValues; // ← valid card register values
    UInt8   padding[1];
};

// 'attributes' field values

enum
{
    csExclusivelyUsed      = 0x0001,
    csEnableIREQs         = 0x0002,
    csVccChangeValid      = 0x0004,
    csVpp1ChangeValid     = 0x0008,
    csVpp2ChangeValid     = 0x0010,
    csValidClient         = 0x0020,
    // request that power be applied to socket during sleep
    csSleepPower          = 0x0040,
```

PC Card Services

```

        csLockSocket          = 0x0080,
        csTurnOnInUse        = 0x0100
};

// 'intType' field values
enum
{
    csMemoryInterface          = 0x01,
    csMemory_And_IO_Interface = 0x02
};

// 'present' field values

enum
{
    csOptionRegisterPresent    = 0x01,
    csStatusRegisterPresent   = 0x02,
    csPinReplacementRegisterPresent = 0x04,
    csCopyRegisterPresent     = 0x08
};

// 'cardValues' field values

enum
{
    csOptionValueValid        = 0x01,
    csStatusValueValid        = 0x02,
    csPinReplacementValueValid = 0x04,
    csCopyValueValid          = 0x08
};

```

DESCRIPTION

The `CSGetConfigurationInfo` function is generally called after a client has parsed a tuple stream, identified an inserted card as its card, and is ready to initialize a `CSModifyConfiguration` parameter block.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid <code>clientHandle</code> value

CSRequestConfiguration

You can use the `CSRequestConfiguration` function to establish yourself as the configuring client for a card and socket and to lock the configuration.

```
pascal OSErr
    CSRequestConfiguration(GetModRequestConfigInfoPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt16 socket;      // → logical socket number
    UInt16 attributes;  // → bitmap of configuration attributes
    UInt8  vcc;         // → Vcc setting
    UInt8  vpp1;        // → Vpp1 setting
    UInt8  vpp2;        // → Vpp2 setting
    UInt8  intType;     // → interface type (memory or memory+I/O)
    UInt32 configBase; // → card base address of configuration registers
    UInt8  status;     // → card status register setting, if present
    UInt8  pin;        // → card pin register setting, if present
    UInt8  copy;       // → card socket/copy reg. setting, if present
    UInt8  configIndex; // → card option register setting, if present
    UInt8  present;    // → bitmap of which config registers are present
    UInt8  firstDevType; // ← from DeviceID tuple
    UInt8  funcCode;   // ← from FuncID tuple
    UInt8  sysInitMask; // ← from FuncID tuple
    UInt16 manufCode;  // ← from ManufacturerID tuple
    UInt16 manufInfo;  // ← from ManufacturerID tuple
    UInt8  cardValues; // ← valid card register values
    UInt8  padding[1]; //
};
```

For attributes, intType, present, and cardValues field values, see “CSGetConfigurationInfo” beginning on page 149.

DESCRIPTION

The `CSRequestConfiguration` function is used by a client to establish a locked configuration on a socket and its card. A client calls `CSRequestConfiguration` after it has parsed an inserted and ready card and has recognized the card as being usable.

Card Services uses `clientHandle` to lock in the configuration until the same client calls `CSReleaseConfiguration`. Once a socket and card are configured no other client may alter their configuration.

PC Card Services

Configuring a socket and card consists of three operations:

- establishing Vcc and Vpp for the socket
- establishing the socket interface definition (memory only or I/O and memory)
- writing the configuration registers on the card

When Card Services receives a `CARD_INSERTION` and subsequent `CARD_READY` event for a socket, it configures the socket by setting Vcc, Vpp1, and Vpp2 to 5 volts; configuring the interface to be memory only; and issuing `RESET` to the card. Card Services then parses the CIS (card information structure) of the card. Once Card Services has finished parsing the CIS, it issues a `CARD_READY` message to all registered clients. (It has previously delivered a `CARD_INSERTION` message to the same clients.) Even if a client parses and recognizes a card and intends to use the card without altering the configuration, it should call `CSRequestConfiguration` to establish itself as the configuring client.

RESULT CODES

<code>SUCCESS</code>	No error
<code>BAD_HANDLE</code>	Invalid <code>clientHandle</code> value
<code>BAD_SOCKET</code>	Invalid socket number
<code>CONFIGURATION_LOCKED</code>	Another client has already locked a configuration
<code>NO_CARD</code>	No card
<code>OUT_OF_RESOURCE</code>	Card Services lacks enough resources to complete this request
<code>BAD_BASE</code>	Invalid base entered

CSModifyConfiguration

You can use the `CSModifyConfiguration` function to alter the configuration of a socket or card.

```
pascal OSErr CSModifyConfiguration(GetModRequestConfigInfoPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt16 socket;      // → logical socket number
    UInt16 attributes; // → bitmap of configuration attributes
    UInt8 vcc;         // → Vcc setting
    UInt8 vpp1;       // → Vpp1 setting
    UInt8 vpp2;       // → Vpp2 setting
    UInt8 intType;    // → interface type (memory or memory+I/O)
```

PC Card Services

```

    UInt32 configBase;      // → card base address of config registers
    UInt8  status;         // → card status register setting, if present
    UInt8  pin;           // → card pin register setting, if present
    UInt8  copy;          // → card socket/copy reg. setting, if present
    UInt8  configIndex;   // → card option register setting, if present
    UInt8  present;       // → bitmap of which config regs. are present
    UInt8  firstDevType;  // ← from DeviceID tuple
    UInt8  funcCode;      // ← from FuncID tuple
    UInt8  sysInitMask;   // ← from FuncID tuple
    UInt16 manifCode;     // ← from ManufacturerID tuple
    UInt16 manifInfo;     // ← from ManufacturerID tuple
    UInt8  cardValues;    // ← valid card register values
    UInt8  padding[1];    //
};

```

For attributes, intType, present, and cardValues field values, see “CSGetConfigurationInfo” beginning on page 149.

DESCRIPTION

The CSModifyConfiguration function is used by clients to alter any of the three configuration elements of a socket or card. Only a client that has previously succeeded in calling CSRequestConfiguration may call CSModifyConfiguration.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid clientHandle value
BAD_SOCKET	Invalid socket number
CONFIGURATION_LOCKED	Another client has already locked a configuration
NO_CARD	No card
OUT_OF_RESOURCE	Card Services lacks enough resources to complete this request
BAD_BASE	Invalid base entered

CSReleaseConfiguration

You can use the CSReleaseConfiguration function to release a previously locked configuration.

```
pascal OSErr CSReleaseConfiguration(ReleaseConfigurationPB *pb);
```

PC Card Services

The parameter block associated with this function is as follows:

```
typedef struct ReleaseConfigurationPB ReleaseConfigurationPB;
struct ReleaseConfigurationPB
{
    UInt32  clientHandle;
    UInt16  socket;
};
```

DESCRIPTION

The CSReleaseConfiguration function is used by clients to release a configuration previously locked for a socket and card.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid clientHandle value
BAD_SOCKET	Invalid socket number
CONFIGURATION_LOCKED	Another client has already locked a configuration
NO_CARD	No card in specified socket

CSAccessConfigurationRegister

You can use the CSAccessConfigurationRegister function to modify a single configuration register. This function is not normally used by clients.

```
pascal OSErr
CSAccessConfigurationRegister(AccessConfigurationRegisterPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct AccessConfigurationRegisterPB
AccessConfigurationRegisterPB;

struct AccessConfigurationRegisterPB
{
    UInt16 socket;           // → global socket number
    UInt8  action;          // → read/write
    UInt8  offset;          // → offset from config register base
    UInt8  value;           // ↔ value to read/write
    UInt8  padding[1];
};
```

PC Card Services

```
// 'action' field values
enum {
    CS_ReadConfigRegister = 0x00,
    CS_WriteConfigRegister = 0x01
};
```

DESCRIPTION

The `CSAccessConfigurationRegister` function lets a client modify a single configuration register. The location of the register is defined by adding `AccessConfigurationRegisterPB.offset` to the configuration base address (see `CSModifyConfiguration` on page 152). If the `action` parameter is set to `CS_ReadConfigRegister`, then the configuration register value is returned in `AccessConfigurationRegisterPB.value`. If the `action` parameter is set to `CS_WriteConfigRegister`, then the configuration register is written with `AccessConfigurationRegisterPB.value`.

IMPORTANT

The `CSAccessConfigurationRegister` function is not normally used by clients. When clients want to set configuration registers they usually call `CSRequestConfiguration` or `CSModifyConfiguration` and set the appropriate registers at that time. ▲

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number

Masks

The functions described in this section get and set client event and socket masks.

CSGetClientEventMask

You can use the `CSGetClientEventMask` function to obtain your current event mask.

```
pascal OSErr CSGetClientEventMask(GetSetClientEventMaskPB *pb);
```

PC Card Services

The parameter block associated with this function is as follows:

```
typedef struct GetSetClientEventMaskPB GetSetClientEventMaskPB;
struct GetSetClientEventMaskPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  attributes;   // → bitmap of attributes
    UInt16  eventMask;    // ← bitmap of events to be passed to
                        //      client for this socket
    UInt16  socket;       // → logical socket number
};

// 'attributes' field values

enum
{
    csEventMaskThisSocketOnly = 0x0001
};

// 'eventMask' field values

enum
{
    csWriteProtectEvent      = 0x0001,
    csCardLockChangeEvent    = 0x0002,
    csEjectRequestEvent      = 0x0004,
    csInsertRequestEvent     = 0x0008,
    csBatteryDeadEvent       = 0x0010,
    csBatteryLowEvent        = 0x0020,
    csReadyChangeEvent       = 0x0040,
    csCardDetectChangeEvent  = 0x0080,
    csPMChangeEvent          = 0x0100,
    csResetEvent             = 0x0200,
    csSSUpdateEvent         = 0x0400,
    csFunctionInterrupt      = 0x0800,
    csAllEvents              = 0xFFFF
};
```

DESCRIPTION

The `CSGetClientEventMask` function is used by a client to obtain its current event mask. If the `GetSetClientEventMaskPB.attributes` field has `csEventMaskThisSocketOnly` reset, the `CSGetClientEventMask` function returns the client's global event mask. If `GetSetClientEventMaskPB.attributes` has `csEventMaskThisSocketOnly` set, then the event mask for the given socket number is returned.

PC Card Services

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid <code>clientHandle</code> value
BAD_SOCKET	Invalid socket number

CSSetClientEventMask

You can use the `CSSetClientEventMask` function to establish your event mask.

```
pascal OSErr CSSetClientEventMask(GetSetClientEventMaskPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetSetClientEventMaskPB GetSetClientEventMaskPB;
struct GetSetClientEventMaskPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  attributes;   // → bitmap of attributes
    UInt16  eventMask;    // → bitmap of events to pass to client
                                //      for this socket
    UInt16  socket;       // → logical socket number
};
```

For `eventMask` field values, see “`CSGetClientEventMask`” on page 155.

DESCRIPTION

The `CSSetClientEventMask` function is used by a client to establish its event mask. If the `GetSetClientEventMaskPB.attributes` field is reset, `CSSetClientEventMask` sets the client’s global event mask. If the `GetSetClientEventMaskPB.attributes` field has `csEventMaskThisSocketOnly` set, then the event mask for the given socket number is set.

After processing `CARD_READY` and determining that the card is not usable, clients should clear their global event masks so that message processing with the system is streamlined.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	The <code>clientHandle</code> field of <code>GetClientInfoPB</code> is invalid
BAD_SOCKET	Invalid socket number

CSRequestSocketMask

You can use the CSRequestSocketMask function to establish an event mask for a specified socket.

```
pascal OSErr CSRequestSocketMask(ReqRelSocketMaskPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct ReqRelSocketMaskPB ReqRelSocketMaskPB;
struct ReqRelSocketMaskPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  socket;       // → logical socket
    UInt16  eventMask;   // → bitmap of events to pass to client
                                //      for this socket
};
```

For eventMask field values, see “CSGetClientEventMask” on page 155.

DESCRIPTION

The CSRequestSocketMask function is used to establish an event mask for the given socket number.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	The clientHandle field of GetClientInfoPB is invalid

CSReleaseSocketMask

You can use the CSReleaseSocketMask function to clear the event mask for a PC card that you are no longer using.

```
pascal OSErr CSReleaseSocketMask(ReqRelSocketMaskPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct ReqRelSocketMaskPB ReqRelSocketMaskPB;
struct ReqRelSocketMaskPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  socket;       // → logical socket
};
```

PC Card Services

```

    UInt16  eventMask;      // → bitmap of events to pass to client
                          //      for this socket
};

```

For eventMask field values, see “CSGetClientEventMask” on page 155.

DESCRIPTION

The CSReleaseSocketMask function is used to clear the event mask for the specified socket. This is the recommended way for clients to clear socket events when they are not using a particular PC card.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	The clientHandle field of GetClientInfoPB is invalid

Tuples

You can use the functions described in this section to obtain PC card information from the corresponding tuples.

CSGetFirstTuple

You can use the CSGetFirstTuple function to obtain access to the first tuple associated with a particular socket.

```
pascal OSErr CSGetFirstTuple(GetTuplePB *pb);
```

The parameter block associated with this function is as follows:

```

typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
    UInt16 socket;      // → logical socket number
    UInt16 attributes; // → bitmap of attributes
    UInt8  desiredTuple; // → desired tuple code value, or $FF for all
    UInt8  tupleOffset; // → offset into tuple from link byte
    UInt16 flags;      // ↔ reserved for internal use
    UInt32 linkOffset  // ↔ reserved for internal use
    UInt32 cisOffset;  // ↔ reserved for internal use
};

```

PC Card Services

```

union
{
    struct
    {
        UInt8  tupleCode;    // ← tuple code found
        UInt8  tupleLink;   // ← link value for tuple found
    } TuplePB;

    struct
    {
        UInt16  tupleDataMax; // → maximum size of tuple data area
        UInt16  tupleDataLen; // ← number of bytes in tuple body
        TupleBody tupleData;  // ← tuple data
    } TupleDataPB;
    } u;
};

// 'attributes' field values
enum
{
    csReturnLinkTuples = 0x0001
};

```

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
IN_USE	Card is configured and being used by another client
READ_FAILURE	Card cannot be read
BAD_CIS	Card Services has encountered a bad CIS structure
OUT_OF_RESOURCE	Card Services is not able to obtain resources to complete
NO_MORE_ITEMS	There are no more tuples to process

CSGetNextTuple

You can use the `CSGetNextTuple` function to obtain access to each tuple associated with a particular socket after you have used the `CSGetFirstTuple` function to obtain access to the first tuple associated with that socket.

```
pascal OSErr CSGetNextTuple(GetTuplePB *pb);
```

PC Card Services

The parameter block associated with this function is as follows:

```
typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
    UInt16 socket;        // → logical socket number
    UInt16 attributes;   // → bitmap of attributes
    UInt8  desiredTuple; // → desired tuple code value, or $FF for all
    UInt8  tupleOffset;  // → offset into tuple from link byte
    UInt16 flags;        // ↔ reserved for internal use
    UInt32 linkOffset;   // ↔ reserved for internal use
    UInt32 cisOffset;    // ↔ reserved for internal use

    union
    {
        struct
        {
            UInt8  tupleCode; // ← tuple code found
            UInt8  tupleLink; // ← link value for tuple found
        } TuplePB;

        struct
        {
            UInt16  tupleDataMax; // → maximum size of tuple data area
            UInt16  tupleDataLen; // ← number of bytes in tuple body
            TupleBody tupleData;   // ← tuple data
        } TupleDataPB;
    } u;
};
```

For attributes field values, see “CSGetFirstTuple” on page 159.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
IN_USE	Card is configured and being used by another client
READ_FAILURE	Card cannot be read
BAD_CIS	Card Services has encountered a bad CIS structure
OUT_OF_RESOURCE	Card Services is not able to obtain resources to complete function
NO_MORE_ITEMS	There are no more tuples to process

CSGetTupleData

You can use the CSGetTupleData function to obtain information for the tuple previously found using either the CSGetNextTuple or CSGetFirstTuple function.

```
pascal OSErr CSGetTupleData(GetTuplePB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
    UInt16 socket;          // → logical socket number
    UInt16 attributes;     // → bitmap of attributes
    UInt8  desiredTuple;   // → desired tuple code value, or $FF for all
    UInt8  tupleOffset;    // → offset into tuple from link byte
    UInt16 flags;          // ↔ internal use
    UInt32 linkOffset;     // ↔ internal use
    UInt32 cisOffset;      // ↔ internal use

    union
    {
        struct
        {
            UInt8  tupleCode;    // ← tuple code found
            UInt8  tupleLink;    // ← link value for tuple found
        } TuplePB;

        struct
        {
            UInt16 tupleDataMax; // → maximum size of tuple data area
            UInt16 tupleDataLen; // ← number of bytes in tuple body
            TupleBody tupleData; // ← tuple data
        } TupleDataPB;
    } u;
};

// 'attributes' field values
enum
{
    csReturnLinkTuples = 0x0001
};
```

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
OUT_OF_RESOURCE	Card Services is not able to obtain resources to complete function

Card and Socket Status

The CSGetStatus function gets card and socket status information.

CSGetStatus

You can use the CSGetStatus function to get status information for the specified socket.

```
pascal OSErr CSGetStatus(GetStatusPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetStatusPB GetStatusPB;

struct GetStatusPB
{
    UInt16 socket;      // → logical socket number
    UInt16 cardState;  // ← current state of installed card
    UInt16 socketState; // ← current state of the socket
};

// 'cardState' field values

enum
{
    csWriteProtected = 0x0001,
    csCardLocked     = 0x0002,
    csEjectRequest   = 0x0004,
    csInsertRequest  = 0x0008,
    csBatteryDead    = 0x0010,
    csBatteryLow     = 0x0020,
    csReady          = 0x0040,
    csCardDetected   = 0x0080
};

// 'socketState' field values
```

PC Card Services

```
enum
{
    csWriteProtectChanged    = 0x0001,
    csCardLockChanged        = 0x0002,
    csEjectRequestPending    = 0x0004,
    csInsertRequestPending   = 0x0008,
    csBatteryDeadChanged     = 0x0010,
    csBatteryLowChanged      = 0x0020,
    csReadyChanged           = 0x0040,
    csCardDetectChanged      = 0x0080
};
```

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number

Access Window Management

The functions described in this section help you manage access windows.

CSRequestWindow

You can use the CSRequestWindow function to establish a new access window.

```
pascal OSErr CSRequestWindow(ReqModRelWindowPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt32 windowHandle; // ↔ window descriptor
    UInt16 socket;       // → logical socket number
    UInt16 attributes;  // → window attributes (bitmap)
    UInt32 base;        // ↔ system base address
    UInt32 size;        // ↔ memory window size
    UInt8  accessSpeed; // → window access speed (bitmap)
                //      (not applicable for I/O mode)
    UInt8  padding[1];
};

// 'attributes' field values
```

PC Card Services

```

enum
{
    csMemoryWindow      = 0x0001,
    csIOWindow          = 0x0002,
    csAttributeWindow   = 0x0004, // not normally used by Card Services
                                // clients
    csWindowTypeMask    = 0x0007,
    csEnableWindow      = 0x0008,
    csAccessSpeedValid  = 0x0010,
    csLittleEndian      = 0x0020, // configure socket for
                                // little-endianness
    cs16BitDataPath     = 0x0040,
    csWindowPaged       = 0x0080,
    csWindowShared      = 0x0100,
    csWindowFirstShared = 0x0200,
    csWindowProgrammable = 0x0400
};

// 'accessSpeed' field values

enum
{
    csDeviceSpeedCodeMask = 0x07,
    csSpeedExponentMask   = 0x07,
    csSpeedMantissaMask   = 0x78,
    csUseWait              = 0x80,

    csAccessSpeed250nsec  = 0x01,
    csAccessSpeed200nsec  = 0x02,
    csAccessSpeed150nsec  = 0x03,
    csAccessSpeed100nsec  = 0x04,

    csExtAccSpeedMant1pt0 = 0x01,
    csExtAccSpeedMant1pt2 = 0x02,
    csExtAccSpeedMant1pt3 = 0x03,
    csExtAccSpeedMant1pt5 = 0x04,
    csExtAccSpeedMant2pt0 = 0x05,
    csExtAccSpeedMant2pt5 = 0x06,
    csExtAccSpeedMant3pt0 = 0x07,
    csExtAccSpeedMant3pt5 = 0x08,
    csExtAccSpeedMant4pt0 = 0x09,
    csExtAccSpeedMant4pt5 = 0x0A,
    csExtAccSpeedMant5pt0 = 0x0B,
    csExtAccSpeedMant5pt5 = 0x0C,
    csExtAccSpeedMant6pt0 = 0x0D,
    csExtAccSpeedMant7pt0 = 0x0E,
    csExtAccSpeedMant8pt0 = 0x0F,

```

PC Card Services

```

    csExtAccSpeedExp1ns    = 0x00,
    csExtAccSpeedExp10ns  = 0x01,
    csExtAccSpeedExp100ns = 0x02,
    csExtAccSpeedExp1000ns = 0x03,
    csExtAccSpeedExp10000ns = 0x04,
    csExtAccSpeedExp100000ns = 0x05,
    csExtAccSpeedExp1000000ns = 0x06,
    csExtAccSpeedExp10000000ns = 0x07
};

```

DIVERGENCE FROM PCMCIA STANDARD

Apple has added another attribute (`csIOTypeWindow`) that lets a client request that its new access window be an I/O cycle window. For an I/O cycle window, speed characteristics are fixed and any speed-related parameters are ignored. Speed parameters are only effective if the access window is of type `Memory` or `Attribute`.

In the PCMCIA standard, there is an implied window assignment when a client calls `CSRequestConfiguration` because the client must have called `RequestI/O` first. This assures the client that there is I/O cycle window support for the change.

RESULT CODES

<code>SUCCESS</code>	No error
<code>BAD_SOCKET</code>	Invalid socket number
<code>OUT_OF_RESOURCE</code>	Card Services is unable to obtain resources to complete function
<code>BAD_BASE</code>	Invalid base address
<code>BAD_ATTRIBUTE</code>	Invalid window attributes

CSModifyWindow

You can use the `CSModifyWindow` function to modify information about an access window.

```
pascal OSErr CSModifyWindow(ReqModRelWindowPB *pb);
```

The parameter block associated with this function is as follows:

```

typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt32 windowHandle; // ↔ window descriptor
    UInt16 socket;       // → logical socket number
    UInt16 attributes;  // → window attributes (bitmap)
    UInt32 base;        // ↔ system base address
};

```

PC Card Services

```

    UInt32 size;           // ↔ memory window size
    UInt8  accessSpeed;   // → window access speed (bitmap)
                          //      (not applicable for I/O mode)

    UInt8  padding[1];
};

```

For attributes and accessSpeed field values, see “CSRequestWindow” on page 164.

DIVERGENCE FROM PCMCIA STANDARD

The CSModifyWindow function must have a valid clientHandle value (the one passed in on CSRequestWindow); otherwise a BAD_HANDLE error is returned.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
OUT_OF_RESOURCE	Card Services is unable to obtain resources to complete function
BAD_BASE	Invalid base address
BAD_ATTRIBUTE	Invalid window attributes
BAD_HANDLE	invalid clientHandle value

CSReleaseWindow

You can use the CSReleaseWindow function to clear an access window that is not longer needed.

```
pascal OSErr CSReleaseWindow(ReqModRelWindowPB *pb);
```

The parameter block associated with this function is as follows:

```

typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt32 windowHandle; // → window descriptor
    UInt16 socket;       // → logical socket number
    UInt16 attributes;  // not used
    UInt32 size;        // not used
    UInt8  accessSpeed; // not used
    UInt8  padding[1];  // not used
};

```

For attributes and accessSpeed field values, see “CSRequestWindow” on page 164.

DIVERGENCE FROM PCMCIA STANDARD

The CSReleaseWindow function must have a valid clientHandle value (the one passed in on CSRequestWindow); otherwise a BAD_HANDLE error is returned.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
BAD_HANDLE	invalid clientHandle value

Client Registration

The functions described in this section help you get information about Card Services and register and deregister clients.

CSGetCardServicesInfo

You can use the CSGetCardServicesInfo function to get information from the Card Services software about the PC cards currently installed.

```
pascal OSErr CSGetCardServicesInfo(GetCardServicesInfoPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct GetCardServicesInfoPB GetCardServicesInfoPB;
struct GetCardServicesInfoPB
{
    UInt8  signature[2];    // ← two ASCII chars 'CS'
    UInt16 count;          // ← total number of sockets installed
    UInt16 revision;       // ← BCD
    UInt16 csLevel;        // ← BCD
    UInt16 reserved;       // → zero
    UInt16 vStrLen;        // ↔ in: client's buffer size
                                // out: vendor string length
    UInt8  *vendorString;  // ↔ in: pointer to buffer to hold CS vendor
                                // string (zero-terminated)
                                // out: CS vendor string copied to buffer
};
```

RESULT CODES

SUCCESS	No error
---------	----------

CSRegisterClient

You can use the `CSRegisterClient` function to register yourself as a client of the Card Services software.

```
pascal OSErr CSRegisterClient(RegisterClientPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct RegisterClientPB RegisterClientPB;
struct RegisterClientPB
{
    UInt32          clientHandle; // ← client descriptor
    PCCardCSClientUPPclientEntry; // → UPP to client's event handler
    UInt16          attributes;   // → bitmap of client attributes
    UInt16          eventMask;    // → bitmap of events to notify client
    Ptr             clientData;   // → pointer to client's data
    UInt16          version;      // → Card Services version
                                //      client expects
};

// 'attributes' field values (see GetClientInfo)

// csMemoryClient          = 0x0001,
// csIOClient              = 0x0004,
// csShareableCardInsertEvents= 0x0008,
// csExclusiveCardInsertEvents= 0x0010
```

DESCRIPTION

Observe these cautions when using `CSRegisterClient`:

- It must not be called at interrupt time.
- You must specify the type of client for event notification order.
- You must set the event mask for types of events client is interested in. The event mask passed in during this call will be set for the global mask and all socket event masks.

DIVERGENCE FROM PCMCIA STANDARD

The `CSRegisterClient` function is synchronous. On returning from `CSRegisterClient`, the `clientHandle` field is valid. Once this call is successful, all clients are expected to support reentrancy. After `CSRegisterClient`, clients normally call `CSVendorSpecific` with `vsCode` set to `vsEnableSocketEvents`.

PC Card Services

RESULT CODES

SUCCESS	No error
OUT_OF_RESOURCE	Card Services is unable to obtain resources to complete function
BAD_ATTRIBUTE	Invalid window attributes

CSDeregisterClient

You can use the `CSDeregisterClient` function to clear client information previously registered with the Card Services software.

```
pascal OSErr CSDeregisterClient(RegisterClientPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct RegisterClientPB RegisterClientPB;
struct RegisterClientPB
{
    UInt32          clientHandle;    // ← client descriptor
    PCardCSCClientUPP clientEntry;  // → UPP to client's event handler
    UInt16          attributes;     // → bitmap of client attributes
    UInt16          eventMask;     // → bitmap of events to notify
                                     // client
    Ptr             clientData;     // → pointer to client's data
    UInt16          version;       // → Card Services version
                                     // client expects
};
```

For attributes field values, see “CSRegisterClient” on page 169.

RESULT CODES

SUCCESS	No error
BAD_ATTRIBUTE	Invalid window attributes
BAD_HANDLE	Invalid clientHandle value

Miscellaneous Functions

The functions described in this section help you with various Card Services management tasks.

CSResetCard

You can use the CSResetCard function to reset a PC card in a specified socket.

```
pascal OSErr CSResetCard(ResetCardPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct ResetCardPB ResetCardPB;
struct ResetCardPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  socket;      // → socket number
    UInt16  attributes;  // not used
};
```

DESCRIPTION

Calling clients will receive RESET_COMPLETE messages regardless of whether or not their socket event mask and global event mask have csResetEvent set.

DIVERGENCE FROM PCMCIA STANDARD

Card Services does not issue CARD_RESET in place of CARD_READY. If a client is issuing a reset to a card, then it should know whether the card will generate a CARD_READY or not. If the card transitions from BSY to RDY, then the client will also know that it shouldn't access the card until it receives the CARD_READY event.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
BAD_HANDLE	Invalid clientHandle value or clientHandle does not match configuring clientHandle

CSValidateCIS

You can use the CSValidateCIS function to find out whether a socket has a valid CIS.

```
pascal OSErr CSValidateCIS(ValidateCISPB *pb);
```

PC Card Services

The parameter block associated with this function is as follows:

```
typedef struct ValidateCISPB ValidateCISPB;
struct ValidateCISPB
{
    UInt16  socket;      // → socket number
    UInt16  chains;     // → whether link/null tuples should be included
};
```

DIVERGENCE FROM PCMCIA STANDARD

The PCMCIA standard specifies that a BAD_CIS result is to be returned by setting the pb->chains element to 0. To accommodate cards that don't have any tuples, Card Services uses the result code to return BAD_CIS (if the CIS is bad). If SUCCESS is returned, then the value in pb->chains reflects the number of valid tuples, with link tuples not counted.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
BAD_CIS	Card Services has detected a bad CIS

CSVendorSpecific

You can use the CSVendorSpecific function to perform certain elements that are Mac OS specific.

```
pascal OSErr CSVendorSpecific(VendorSpecificPB *pb);
```

The parameter block associated with this function is as follows:

```
typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  vsCode;
    UInt16  socket;
    UInt32  dataLen;      // → length of buffer pointed to by vsDataPtr
    UInt8   *vsDataPtr;  // → Card Services version this client expects
};

// 'vsCode' field values
```

PC Card Services

```
enum
{
    vsAppleReserved          = 0x0000,
    vsEjectCard              = 0x0001,
    vsGetCardInfo            = 0x0002,
    vsEnableSocketEvents     = 0x0003,
    vsGetCardLocationIcon    = 0x0004,
    vsGetCardLocationText    = 0x0005,
    vsGetAdapterInfo         = 0x0006
};
```

DESCRIPTION

The `CSVendorSpecific` function is provided to allow Apple Computer to extend the interface definition of Card Services for elements that are Mac OS specific. This function requires two parameters, `clientHandle` and `vsCode`. For each `vsCode` there may be additional parameters required. The following sections describe the additional parameters required for each `vsCode` selector.

RESULT CODES

SUCCESS	No error
UNSUPPORTED_FUNCTION	The <code>vsCode</code> value is invalid

EjectCard Parameter Block

You can use vendor-specific call #1 to eject a card.

```
// vendor-specific call #1
```

The parameter block associated with this function is as follows:

```
typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  vsCode;      // → vsCode = 1
    UInt16  socket;      // → desired socket number to eject
    UInt32  dataLen;     // not used
    UInt8   *vsDataPtr;  // not used
};
```

PC Card Services

DESCRIPTION

Clients must pass in their `clientHandle` value to eject cards that they have configured. Clients may not be able to eject cards that they did not configure unless the card is previously unconfigured.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket
IN_USE	Another client refused the ejection request

GetCardInfo Parameter Block

You can use vendor-specific call #2 to get information about a card in a socket.

```
// vendor-specific call #2
```

The parameter block associated with this function is as follows:

```
typedef struct GetCardInfoPB GetCardInfoPB;
struct GetCardInfoPB
{
    UInt8  cardType;    // ← type of card in socket
                    //      (defined at top of file)
    UInt8  subType;    // ← detailed card type (defined at top of file)
    UInt16 reserved;  // ↔ reserved (should be set to 0)
    UInt16 cardNameLen; // → maximum length of card name to be returned
    UInt16 vendorNameLen; // → max. length of vendor name to be returned
    UInt8  *cardName;  // → ptr to card name string (from CIS), or nil
    UInt8  *vendorName; // → ptr to vendor name (from CIS), or nil
};

// GetCardInfo card types

#define  csUnknownCardType      0
#define  csMultiFunctionCardType 1
#define  csMemoryCardType      2

#define  csSerialPortCardType   3
#define  csSerialOnlyType      0
#define  csDataModemType       1
#define  csFaxModemType        2
#define  csFaxAndDataModemMask (csDataModemType | csFaxModemType)
#define  csVoiceEncodingType   4

#define  csParallelPortCardType 4
```

PC Card Services

```

#define csFixedDiskCardType      5
#define csUnknownFixedDiskType  0
#define csATAInterface           1
#define csRotatingDevice         (0<<7)
#define csSiliconDevice          (1<<7)
#define csVideoAdaptorCardType   6

#define csNetworkAdaptorCardType 7
#define csAIMSCardType           8
#define csNumCardTypes           9

```

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number
NO_CARD	No card in specified socket

EnableSocketEvents Parameter Block

You can use vendor-specific call #3 to enable events on every socket in the system.

```
// vendor-specific call #3
```

The parameter block associated with this function is as follows:

```

typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
    UInt32 clientHandle; // → clientHandle returned by RegisterClient
    UInt16 vsCode;       // → vsCode = 3
    UInt16 socket;       // not used
    UInt32 dataLen;      // not used
    UInt8 *vsDataPtr;   // not used
};

```

DESCRIPTION

Calling this function is like calling the `CSRequestSocketMask` function for every socket in the system, using the global event mask as the starting socket event mask.

DIVERGENCE FROM PCMCIA STANDARD

This function is not in the PCMCIA specification. After reentrancy into a client is available, calling this function to enable events is better than making repeated calls to the `RequestSocketMask` function.

RESULT CODES

SUCCESS	No error
BAD_HANDLE	Invalid clientHandle value

GetAdapterInfo Parameter Block

You can use vendor-specific call #6 to get information about an adapter that interfaces to a specified socket.

```
// vendor-specific call #6
```

The parameter block associated with this function is as follows:

```
typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
    UInt32  clientHandle; // → clientHandle returned by RegisterClient
    UInt16  vsCode;       // → vsCode = 6
    UInt16  socket;       // → socket number
    UInt32  dataLen;      // → length of GetAdapterInfoPB plus space for
                        //      voltages
    UInt8   *vsDataPtr;   // → GetAdapterInfoPB * (supplied by client)
};

typedef struct GetAdapterInfoPB GetAdapterInfoPB;

struct GetAdapterInfoPB
{
    UInt32  attributes;   // ← capabilities of socket's adapter
    UInt16  revision;    // ← revision ID of adapter
    UInt16  reserved;    //
    UInt16  numVoltEntries; // ← number of valid voltage values
    UInt8   *voltages;   // ←-> array of BCD voltage values
};

// 'attributes' field values

enum
{
    csLevelModeInterrupts      = 0x00000001,
    csPulseModeInterrupts     = 0x00000002,
    csProgrammableWindowAddr   = 0x00000004,
    csProgrammableWindowSize  = 0x00000008,
    csSocketSleepPower         = 0x00000010,
```

PC Card Services

```

    csSoftwareEject          = 0x00000020,
    csLockableSocket        = 0x00000040,
    csInUseIndicator        = 0x00000080
};

```

DESCRIPTION

There are many instances where Socket Services API elements are not brought out to the Card Services API but the elements are required for normal card operation. This call allows clients to query the capabilities of an adapter that interfaces to a given socket. This information may be used to improve the operation of a client with a given socket and card.

RESULT CODES

SUCCESS	No error
BAD_SOCKET	Invalid socket number

CSRequestExclusive and CSReleaseExclusive

The functions `CSRequestExclusive` and `CSReleaseExclusive` are not supported by the Macintosh PowerBook Card Services software.

PC Card Manager Constants

This section lists all the constants used by the PC Card Manager.

```

// miscellaneous

#define CS_MAX_SOCKETS 32 // a long is used as a socket bitmap

enum
{
    gestaltCardServicesAttr = 'pccd', // Card Services attributes
    gestaltCardServicesPresent = 0 // if set, Card Services is
present
};

enum
{
    _PCCardDispatch = 0xAAF0 // Card Services entry trap
};

```

PC Card Services

```

/*
   The PC Card Manager will migrate toward a complete Macintosh name
   space very soon. Part of that process will be to reassign result codes
   to a range reserved for the PC Card Manager. The range will be -9050 to
   -9305 (decimal inclusive).
*/

// result codes
enum
{
    SUCCESS           = 0x00,    // request succeeded
    BAD_ADAPTER      = 0x01,    // invalid adapter number
    BAD_ATTRIBUTE    = 0x02,    // attributes field value is invalid
    BAD_BASE         = 0x03,    // base system memory address is invalid
    BAD_EDC          = 0x04,    // EDC generator specified is invalid
    RESERVED_5      = 0x05,    // «reserved for historical purposes»
    BAD_IRQ          = 0x06,    // specified IRQ level is invalid
    BAD_OFFSET      = 0x07,    // PC card memory array offset is invalid
    BAD_PAGE         = 0x08,    // specified page is invalid
    READ_FAILURE    = 0x09,    // unable to complete read request
    BAD_SIZE        = 0x0A,    // specified size is invalid
    BAD_SOCKET      = 0x0B,    // specified physical socket number is invalid
    RESERVED_C      = 0x0C,    // «reserved for historical purposes»
    BAD_TYPE        = 0x0D,    // window or interface type is invalid
    BAD_VCC         = 0x0E,    // Vcc power level index is invalid
    BAD_VPP         = 0x0F,    // Vpp1 or Vpp2 power level index is invalid
    RESERVED_10     = 0x10,    // «reserved for historical purposes»
    BAD_WINDOW      = 0x11,    // specified window is invalid
    WRITE_FAILURE   = 0x12,    // unable to complete write request
    RESERVED_13     = 0x13,    // «reserved for historical purposes»
    NO_CARD         = 0x14,    // no PC card in the socket
    UNSUPPORTED_FUNCTION= 0x15, // not supported by this implementation
    UNSUPPORTED_MODE = 0x16,    // mode is not supported
    BAD_SPEED       = 0x17,    // specified speed is unavailable
    BUSY            = 0x18,    // unable to process request at this time
    GENERAL_FAILURE = 0x19,    // an undefined error has occurred
    WRITE_PROTECTED = 0x1A,    // media is write protected
    BAD_ARG_LENGTH  = 0x1B,    // ArgLength argument is invalid
    BAD_ARGS        = 0x1C,    // values in argument packet are invalid
    CONFIGURATION_LOCKED= 0x1D, // a configuration has already been locked
    IN_USE          = 0x1E,    // resource is being used by a client
    NO_MORE_ITEMS   = 0x1F,    // there are no more of the requested item
    OUT_OF_RESOURCE = 0x20,    // Card Services has exhausted the resource
    BAD_HANDLE      = 0x21,    // clientHandle value is invalid
    BAD_CIS         = 0x22,    // CIS on card is invalid
};

```

PC Card Services

```

// messages sent to client's event handler
enum
{
    NULL_MESSAGE      = 0x00, // no messages pending
                          // (not sent to clients)

    CARD_INSERTION    = 0x01, // card has been inserted into the socket
    CARD_REMOVAL      = 0x02, // card has been removed from the socket
    CARD_LOCK         = 0x03, // card is locked into the socket with
                          // a mechanical latch
    CARD_UNLOCK       = 0x04, // card is no longer locked into the socket
    CARD_READY        = 0x05, // card is ready to be accessed
    CARD_RESET        = 0x06, // physical reset has completed
    INSERTION_REQUEST = 0x07, // request to insert a card using
                          // insertion motor
    INSERTION_COMPLETE = 0x08, // insertion motor has finished
                          // inserting
                          // a card
    EJECTION_REQUEST  = 0x09, // user or other client is requesting a
                          // card ejection
    EJECTION_FAILED   = 0x0A, // eject failure due to electrical or
                          // mechanical problems
    PM_RESUME         = 0x0B, // power management resume (TBD)
    PM_SUSPEND        = 0x0C, // power management suspend (TBD)
    EXCLUSIVE_REQUEST = 0x0D, // client is trying to obtain exclusive
                          // card access
    EXCLUSIVE_COMPLETE = 0x0E, // indicates whether or not
                          // RequestExclusive succeeded
    RESET_PHYSICAL    = 0x0F, // physical reset is about to occur
    RESET_REQUEST     = 0x10, // client has requested physical reset
    RESET_COMPLETE    = 0x11, // ResetCard() background reset has
                          // completed
    BATTERY_DEAD      = 0x12, // battery is no longer usable;
                          // data will be lost
    BATTERY_LOW       = 0x13, // battery is weak and should
                          // be replaced
    WRITE_PROTECT     = 0x14, // card is now write protected
    WRITE_ENABLED     = 0x15, // card is now write enabled
    ERASE_COMPLETE    = 0x16, // queued background erase request
                          // has completed
    CLIENT_INFO       = 0x17, // client is to return
                          // client information
    SS_UPDATED        = 0x18, // AddSocketServices/ReplaceSocket
                          // services has changed SS support
    FUNCTION_INTERRUPT = 0x19, // card function interrupt
}

```

PC Card Services

```
ACCESS_ERROR      = 0x1A,    // client bus errored on access
                    //      to socket
CARD_UNCONFIGURED = 0x1B,    // a CARD_READY was delivered to all
                    //      clients and no client requested
                    //      a configuration for the socket
STATUS_CHANGED    = 0x1C    // status change for cards in I/O mode
};
```

Glossary

ADB See **Apple Desktop Bus**.

APDA Apple Computer's worldwide direct distribution channel for Apple and third-party development tools and documentation products.

API See **application programming interface**.

Apple Desktop Bus (ADB) An asynchronous bus used to connect relatively slow user-input devices to Apple computers.

AppleTalk Apple Computer's local area networking protocol.

application programming interface (API) The calls and data structures that allow application software to use the features of the operating system.

Baboon The custom IC that provides the interface to the expansion bay in an Omega computer.

big-endian Data formatting in which each field is addressed by referring to its most significant byte. See also **little-endian**.

blit Block transfer operations, often used in graphics software (from the opcode BLT).

Card Services The part of the Macintosh PC Card Manager that provides system services for control software in PCMCIA cards.

client A device driver or application program that uses the Card Services software.

codec A digital encoder and decoder.

color depth The number of bits required to encode the color of each pixel in a display.

Combo The custom IC that supports the serial I/O port and the external SCSI devices in an Omega computer.

DAC See **digital-to-analog converter**.

data burst Multiple longwords of data sent over a bus in a single, uninterrupted stream.

digital-to-analog converter (DAC) A device that produces an analog electrical signal in response to digital data.

direct memory access (DMA) A process for transferring data rapidly into or out of RAM without passing it through a processor or buffer.

DLPI Data Link Provider Interface, the standard networking model used in Open Transport.

DMA See **direct memory access**.

DRAM See **dynamic random-access memory**.

dynamic random-access memory (DRAM) Random-access memory in which each storage address must be periodically interrogated ("refreshed") to maintain its value.

ECSC The enhanced color support chip, a custom IC that provides the data and control interface to the flat panel display.

Ethernet A high-speed local area network technology that includes both cable standards and a series of communications protocols.

GCR See **group code recording**.

global client queue The first-in, first-out queue where the Card Services software keeps information about all its clients.

group code recording (GCR) An Apple recording format for floppy disks.

input/output (I/O) Parts of a computer system that transfer data to or from peripheral devices.

I/O See **input/output**.

little-endian Data formatting in which each field is addressed by referring to its least significant byte. See also **big-endian**.

LocalTalk The cable terminations and other hardware that Apple supplies for local area networking from Macintosh serial ports.

Macintosh PC Exchange An application that runs on Macintosh computers and reads other floppy disk formats, including DOS and ProDOS.

mini-DIN An international standard form of cable connector for peripheral devices.

nonvolatile RAM RAM that retains its contents even when the computer is turned off; also known as parameter RAM.

NuBus A bus architecture in Apple computers that supports plug-in expansion cards.

PC card An expansion card that conforms to the PCMCIA standard.

PC Card Manager The part of Mac OS that supports PC cards in Macintosh PowerBook computers.

PCMCIA standard An industry standard for computer expansion cards.

pixel Contraction of *picture element*; the smallest dot that can be drawn on a display.

Power Manager IC A 68HC05 microprocessor that provides several utility functions in the Omega computer, including ADB support, power control, and keyboard scanning.

Pratt The custom IC that provides memory control and also acts as the bridge between the microprocessor bus on the secondary logic board and the I/O bus on the main logic board.

SCC See **Serial Communications Controller**.

SCSI See **Small Computer System Interface**.

Serial Communications Controller (SCC) Circuitry on the Curio IC that provides an interface to the serial data ports.

Small Computer System Interface (SCSI) An industry standard parallel bus protocol for connecting computers to peripheral devices such as hard disk drives.

socket The hardware receptacle into which a PC Card is inserted.

Socket Services The layer of software that is responsible for communication between Card Services and the socket controller hardware.

TREX The custom IC that provides the interface and control signals for the PCMCIA slots in the Omega computer.

tuple A parsable data group containing configuration information for a PCMCIA card.

Versatile Interface Adapter (VIA) The interface for system interrupts that is standard on most Apple computers.

VIA See **Versatile Interface Adapter**.

video RAM (VRAM) Random-access memory used to store both static graphics and video frames.

VRAM See **video RAM**.

Index

Numerals

68HC05 microprocessor 13

A

AC adapter 6
access to internal components 24
access windows 164
active matrix display 25
ADB connector 28
ADB port 28
appearance 3
Apple Desktop Bus. *See* ADB
Ariel CLUT-DAC IC 15
ATA_Abort function 105
ATA_BusInquiry function 106
ATA disk driver 81, 83–98
 control functions 86–93
 Control routine 84
 Device Manager routines 84–85
 driverGestalt parameter block 94
 status functions 93–98
 Status routine 85
ATA disk driver functions
 clear partition mounting 90
 clear partition write protect 91
 driver gestalt 94
 drive status 94
 eject 87
 format 86
 get a drive 92
 get boot partition 96
 get drive icon 87
 get drive information 88
 get media icon 88
 get partition information 97
 get partition mount status 96
 get partition write protect status 97
 get power mode 98
 mount volume 92
 register partition 91
 set partition mounting 89
 set partition write protect 90
 set power mode 93
 set startup partition 89
 verify 86

ATA_DrvrDeregister function 108
ATA_DrvrRegister function 109
ATA_EjectDrive function 112
ATA_ExecIO function 113
ATA_FindRefNum function 116
ATA_GetDevConfig function 117
ATA_GetDevLocationIcon function 120
ATA hard disk drives, compared with SCSI drives 81
ATA_Identify function 121
ATA/IDE specification 80
ATA interface 18
ATA Manager 98–139
 making calls to 99
 purpose of 81, 83
ATA Manager functions
 ATA_Abort 105
 ATA_BusInquiry 106
 ATA_DrvrDeregister 108
 ATA_DrvrRegister 109
 ATA_EjectDrive 112
 ATA_ExecIO 113
 ATA_FindRefNum 116
 ATA_GetDevConfig 117
 ATA_GetDevLocationIcon 120
 ATA_Identify 121
 ATA_MgrInquiry 122
 ATA_NOP 124
 ATA_QRelease 125
 ATA_RegAccess 125
 ATA_ResetBus 129
 ATA_SetDevConfig 130
ATA_MgrInquiry function 122
ATA_NOP function 124
ATA parameter block header 99
ataPBHdr structure 99–104
ATA_QRelease function 125
ATA_RegAccess function 125
ATA_ResetBus function 129
ATA_SetDevConfig function 130
ATA software
 ATA disk driver 81
 ATA Manager 83
 error codes 139

B

Baboon custom IC 14
back view 5
batteries 5, 6

C

Card Services software 144–180

access window functions

CSModifyWindow 166

CSReleaseWindow 167

CSRequestWindow 164

client information functions

CSGetClientInfo 146

CSGetFirstClient 144

CSGetNextClient 145

client registration functions

CSDeregisterClient 170

CSGetCardServicesInfo 168

CSRegisterClient 169

clients 144

configuration functions

CSAccessConfigurationRegister 154

CSGetConfigurationInfo 149

CSModifyConfiguration 152

CSReleaseConfiguration 153

CSRequestConfiguration 151

gestalt constant 177

masking functions

CSGetClientEventMask 155

CSReleaseSocketMask 158

CSRequestSocketMask 158

CSSetClientEventMask 157

messages 179

miscellaneous functions

CSResetCard 171

CSValidateCIS 171

CSVendorSpecific 172

result codes 178

status function, CSGetStatus 163

tuples functions

CSGetFirstTuple 159

CSGetNextTuple 160

CSGetTupleData 162

unsupported functions

CSReleaseExclusive 177

CSRequestExclusive 177

vendor-specific calls

EjectCard 173

EnableSocketEvents 175

GetAdapterInfo 176

GetCardInfo 174

clear partition mounting function 90

clear partition write protect function 91

clients, registration of 168

Combo custom IC 13

compatibility 6

 sound sample rates 7

configurations 5

connectors

 ADB 28

 expansion bay 33

 external video 50, 51

 hard disk 21

 RAM expansion 39, 42

 RAM expansion card 47

 SCSI 26

 serial port 25

 video 50

control functions, of the ATA disk driver 86–93

Control routine 84

CSAccessConfigurationRegister function 154

CSC custom IC 14, 24

CSDeregisterClient function 170

CSGetCardServicesInfo function 168

CSGetClientEventMask function 155

CSGetClientInfo function 146

CSGetConfigurationInfo function 149

CSGetFirstClient function 144

CSGetFirstTuple function 159

CSGetNextClient function 145

CSGetNextTuple function 160

CSGetStatus function 163

CSGetTupleData function 162

CSModifyConfiguration function 152

CSModifyWindow function 166

CSRegisterClient function 169

CSReleaseConfiguration function 153

CSReleaseExclusive function 177

CSReleaseSocketMask function 158

CSReleaseWindow function 167

CSRequestConfiguration function 151

CSRequestExclusive function 177

CSRequestSocketMask function 158

CSRequestWindow function 164

CSResetCard function 171

CSSetClientEventMask function 157

CSValidateCIS function 171

CSVendorSpecific function 172

custom ICs

 Ariel 15

 Baboon 14

 Combo 13

 CSC 14, 24

 Keystone 15

 Pratt 10, 11

 Singer 13, 29

 TREX 15

 Whitney 12

D

Device Manager 73
display controller IC 14
Display Manager 68
displays
 active matrix 24, 25
 backlighting 24
 dual mode 49
 DualScan 25
 external video monitors 48, 49
 adapter cable 50
 flat panel types 24
 FSTN 25
 mirror mode 7, 49
 NuBus card emulation 24
 number of colors 7, 25
 passive matrix 24
 supertwist 24, 25
 TFT 25
driver gestalt function 94
driverGestalt parameter block 94
Drive Setup, modifications to 68
drive status function 94
dual mode 49
DualScan display 25

E

EjectCard vendor-specific call 173
eject function 87
EnableSocketEvents vendor-specific call 175
error codes 139
Ethernet driver 64
event mask 155
expansion bay 32–38
 device installation 38
expansion bay connector 33–37
 signal assignments 34
 signal definitions 35–37
expansion bay controller IC 14
external video port 49

F

features summary 2
Finder modifications for large volume support 67, 70
flat panel displays 24
format function 86
FPU (floating-point unit) 11
front view 4
function-key software 64

G

Gestalt function 75
gestaltMachineType value 62
GetAdapterInfo vendor-specific call 176
get a drive function 92
get boot partition function 96
GetCardInfo vendor-specific call 174
get drive icon function 87
get drive information function 88
get media icon function 88
get partition information function 97
get partition mount status function 96
get partition write protect status function 97
get power mode function 98

H

hard disk 18
 dimensions 18
 IDE data bus 21
hard disk capacity 5
hard disk connector 21
 pin assignments on 21
 signals on 22
HDI-30 connector 26
HFS volume format 70

I, J

IDE disk interface 18
IDE hard disk 18
 connector 20
 data bus 21
 dimensions 18
 signals 22
identifying the computers 62
IDE specification 80
infrared module 29
input/output subsystem 10
 MC68030 bus 11
I/O ports
 SCSI 26
 serial 25
 video 49, 50

K

keyboards 23
 function keys 64
 ISO layout 24
 removing 24
 United States layout 23
 Keystone video timing IC 15

L

large partition support 66
 large volume support 66, 70
 allocation blocks 70
 extended API 66
 extended data structures 71
 extended parameter block 71, 73
 limitations 67
 maximum file size 71
 modified applications 67
 requirements 71

M, N, O

MC68040 microprocessor 11
 MC68LC040 microprocessor 11
 memory controller software 63
 memory expansion 6, 11
 microprocessor
 clock speed 11
 type 11
 mirror mode 7, 49
 monitor sense codes 51
 mount volume function 92

P

PBXGetVolInfo function 75
 PC cards 57, 144
 Finder extension for 57
 software eject 57
 PCMCIA cards. *See* PC cards
 PCMCIA slots 15, 57–59
 access windows 58
 data access modes 58
 features 57
 power 59
 signal definitions 58
 specifications 58–59

peripheral devices 6
 peripheral support IC 12
 pointing device 23
 Power Manager IC 13
 trackpad registers in 65
 Power Manager software 63
 data structures 7
 unsafe assumptions 8
 Pratt custom IC, as bus bridge 12
 Pratt memory controller 11
 processor clock speed 5
 processor/memory subsystem 10, 11

Q

QuickDraw acceleration API 68

R

RAM
 expansion 39–47
 addressing 43
 DRAM devices 44
 RAM banks 44
 signals 39, 42
 expansion card 11
 refresh 12
 size of 5
 RAM expansion 6
 RAM expansion card 39–47
 connector 47
 dimensions 46
 DRAM devices 44
 electrical limits 45
 mechanical design of 46–47
 RAM banks 44
 reference documents xii
 register partition function 91
 ROM
 address range 11
 implementation of 11
 software features 62
 ROM software features 62

S

SCC IC 13
 SCSI controller IC 13
 SCSI port 26

secondary logic board 11
 serial port 25
 set partition mounting function 89
 set partition write protect function 90
 set power mode function 93
 set startup partition function 89
 Singer custom IC 13
 Singer sound IC 29
 68HC05 microprocessor 13
 socket mask 155
 sound circuits 29
 sound features 63
 sound IC 29
 sound sample rates 7
 status functions 93–98
 Status routine 85
 Supertwist display 25
 System 7.5 66

T, U

TFT display 25
 trackball 23
 trackpad 23
 software support for 65
 TREX custom IC 15
 tuple information 83, 159

V

VCB allocation block size 70
 verify function 86
 video adapter cable 50
 video card 15, 48–57
 video connector 50
 video controller IC 15
 video mirror mode 7, 49
 video modes
 dual 49
 mirror 7, 49
 video monitors 48, 49
 adapter cable for 50
 sense codes 51
 VGA and SVGA 51
 video output IC 15
 video port 49

W

Whitney custom IC 12

X, Y, Z

XIOPParam data structure 73
 XVOLUMEParam parameter block 71

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages and final pages were created on an Apple LaserWriter Pro printer. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. 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

Allen Watson

DEVELOPMENTAL EDITORS

Wendy Krafft, Beverly Zegarski

ILLUSTRATOR

Sandee Karr

PRODUCTION EDITOR

Rex Wolf