Developer Note

Macintosh PowerBook 5300 Computer

Macintosh PowerBook 5300/100
Macintosh PowerBook 5300c/100
Macintosh PowerBook 5300cs/100
Macintosh PowerBook 5300ce/117

Developer Press

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 xiii
  Typographical Conventions xiii
  Standard Abbreviations xiv

Chapter 1 Introduction 1

Features 2
Appearance 3
Peripheral Devices 4
Configurations 5
Compatibility Issues 5
  RAM Expansion Cards 5
  Number of Colors 5
  Video Mirror Mode 6
  Sound Sample Rates 6
  Power Manager Interface 6
  Microprocessor Differences 7
    Completion Serialized Instructions 7
    Split Cache 7
    Data Alignment 7
  POWER-Clean Code 8

Chapter 2 Architecture 9

Processor/Memory Subsystem 11
  Main Processor 11
  RAM 11
  ROM 12
  PBX Memory Controller IC 12
    Memory Control 12
    Bus Bridge 12

Input/Output Subsystem 13
  Whitney Peripheral Support IC 13
  Combo IC 14
  Singer IC 14
  Power Manager IC 14
  Display Controller IC 14
  Baboon Custom IC 15
  TREX Custom IC 15
Video Card 16
  Keystone Video Controller IC 16
  Ariel Video Output IC 16

Chapter 3 I/O Features 17

Internal IDE Hard Disk Drive 18
  Hard Disk Specifications 18
  Hard Disk Connector 19
    Connector Location 20
    Signal Assignments 20
    IDE Signal Descriptions 21
Trackpad 22
Keyboard 23
Flat Panel Display 24
  Flat Panel Display Circuitry 24
  Number of Colors 24
Serial Port 25
SCSI Port 25
ADB Port 27
Infrared Module 28
Sound System 28
  Sound Inputs 29
  Sound Outputs 29

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 36
    Unused IDE Signals 37
    Power on the Expansion Bay 38
  User Installation of an Expansion Bay Device 38
    Sequence of Control Signals 38
    Guidelines for Developers 39

RAM Expansion 39
  Electrical Design Guidelines for the RAM Expansion Card 40
    Connector Pin Assignments 40
    Signal Descriptions 42
    Address Multiplexing 43
    Banks of DRAM 45
    DRAM Device Requirements 45
    Expansion Card Electrical Limits 46
  Mechanical Design of the RAM Expansion Card 47
    RAM Card Dimensions 47
    RAM Card Connector 48
Video Card 49
  The Apple Video Card 49
    Monitors Supported 49
    Video Mirroring 50
    External Video Connector 51
    Monitor Sense Codes 52
  Video Card Design Guide 53
    Video Card Connector 53
    Signals on the Video Card Connector 53
    Video Card Mechanical Design 55
PCMCIA Slot 58
  PCMCIA Features 58
  Summary Specifications 59
    Access Windows 59
    Data Access 59
    Signal Definitions 60
    Power 60
    Controller Interrupts 60

Chapter 5 Software Features 61

ROM Software 62
  PowerPC 603 Microprocessor 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 67
Application-Level Software 67
Limitations 68
Drive Setup 68
Improved File Sharing 68
Dynamic Recompilation Emulator 68
Resource Manager in Native Code 69
Math Library 69
New BlockMove Extensions 69
POWER-Clean Native Code 71
POWER Emulation 72
POWER-Clean Code 72
Emulation and Exception Handling 72
Code Fragments and Cache Coherency 72
Limitations of PowerPC 601 Compatibility 73
QuickDraw Acceleration API 73
Display Manager 74

Chapter 6 Large Volume Support 75

Overview of the Large Volume File System 76
API Changes 76
Allocation Block Size 76
File Size Limits 77
Compatibility Requirements 77
The API Modifications 77
Data Structures 77
Extended Volume Parameter Block 77
Extended I/O Parameter Block 79
New Extended Function 81

Chapter 7 Power Manager Interface 85

About the Power Manager Interface 86
Things That May Change 86
Checking for Routines 87
Power Manager Interface Routines 87
Header File for Power Manager Dispatch 107

Chapter 8 **Software for ATA Devices** 115

Introduction to the ATA Software 116
  ATA Disk Driver 117
    Drives on PC Cards 118
    Drives in the Expansion Bay 119
  ATA Manager 119
ATA Disk Driver Reference 120
  Standard Device Routines 120
    The Control Routine 120
    The Status Routine 121
  Control Functions 122
  Status Functions 130
ATA Manager Reference 135
  The ATA Parameter Block 135
  Functions 141
Using the ATA Manager With Drivers 168
  Notification of Device Events 168
  Device Driver Loading 169
    New API Entry Point for Device Drivers 170
    Loading a Driver From the Media 171
    Notify-All Driver Notification 172
    ROM Driver Notification 173
  Device Driver Purging 173
  Setting the I/O Speed 175
Error Code Summary 175

Chapter 9 **PC Card Services** 179

Client Information 180
Configuration 185
Masks 192
Tuples 196
Card and Socket Status 200
Access Window Management 201
Client Registration 205
Miscellaneous Functions 208
PC Card Manager Constants 215

Glossary 219

Index 221

Figures and Tables

Chapter 1 **Introduction** 1

Figure 1-1 Front view of the computer 3
Figure 1-2 Back view of the computer 4

Table 1-1 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 IDE 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, United States layout 23
Figure 3-5 Keyboard, ISO layout 23
Figure 3-6 Serial port connector 25
Figure 3-7 ADB connector 27

Table 3-1 Pin assignments on the IDE hard disk connector 20
Table 3-2 Signals on the IDE hard disk connector 21
Table 3-3 Characteristics of the displays 24
Table 3-4 Serial port signals 26
Table 3-5 SCSI connector signals 26
Table 3-6 ADB connector pin assignments 27

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 47
Figure 4-4 Dimensions of the RAM expansion card 47
Figure 4-5 Restricted areas on the component side of the card 48
Figure 4-6 Video card 49
Figure 4-7 Video connectors 52
Figure 4-8 Dimensions of the video card 55
Figure 4-9 Video card and 80-pin connector 56
Figure 4-10 Video card bottom view with component restrictions 56
Figure 4-11 Video card top view with component restrictions 57
Figure 4-12 Video card top view 57
Figure 4-13 Detail of EMI shield mounting holes 58

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	38
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	50
Table 4-12	Signals on the video connector	51
Table 4-13	Monitor sense codes	52
Table 4-14	Signals on the video card connector	53
Table 4-15	Descriptions of the signals on the video card connector	55

Chapter 5 Software Features 61

Table 5-1	Summary of `BlockMove` routines	70

Chapter 7 Power Manager Interface 85

Table 7-1	Interface routines and their selector values	88

Chapter 8 Software for ATA Devices 115

Figure 8-1	ATA software model	116

Table 8-1	Control functions	120
Table 8-2	Status functions	122
Table 8-3	Control bits in the `ataFlags` field	138
Table 8-4	ATA Manager functions	141
Table 8-5	Event masks	146
Table 8-6	Bits in pcValid field	156
Table 8-7	ATA register selectors	164
Table 8-8	Register mask bits	164
Table 8-9	Event codes send by the ATA Manager	168
Table 8-10	Input parameter bits for the old API	170
Table 8-11	Input parameter bits for the new API	171
Table 8-12	Purge permissions and responses	174
Table 8-13	ATA driver error codes	175

About This Developer Note

This developer note describes the Macintosh PowerBook 5300 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 nine chapters.

- Chapter 1, "Introduction," introduces the Macintosh PowerBook 5300 computer and describes its new features.
- Chapter 2, "Architecture," describes the internal logic of the 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. It includes development guides for the RAM expansion card, the PDS card, and the communications cards.
- 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, "Power Manager Interface," describes the latest revision of the application interface for the Power Manager software.
- Chapter 8, "Software for ATA Devices," describes the low-level program interface used by utility software for the IDE hard disk drive.
- Chapter 9, "PC Card Services," describes a new part of Mac OS that supports software using PC Cards in the PCMCIA slots.

This developer note also contains a glossary and an index.

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 *Guide to the Macintosh Family Hardware,* second edition, *Designing Cards and Drivers for the Macintosh Family,* third edition, and the relevant volumes of *Inside Macintosh.* These Apple books are available in technical bookstores and through APDA.

For information about PC cards and the PCMCIA slot, developers should have a copy of *Developing PC Card Software for the Mac OS.* That book is scheduled for publication at about the time the Macintosh PowerBook 5300 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 may wish to have copies of the *Macintosh Classic II, Macintosh PowerBook Family, and Macintosh Quadra Family Developer Notes;* and *Macintosh Developer Notes,* numbers 1–5 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.

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	800-282-2732 (United States) 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

For information about programming the PowerPC™ family microprocessors, developers should have copies of Motorola's *PowerPC 601 RISC Microprocessor User's Manual* and *PowerPC 603 Microprocessor Implementation Definition Book IV*.

For information about ATA devices such as the built-in IDE hard disk, developers should have access to 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.

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.

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	MHz	megahertz
dB	decibels	mm	millimeters
GB	gigabytes	ms	milliseconds
Hz	hertz	mV	millivolts
K	1024	µF	microfarads
KB	kilobytes	ns	nanoseconds
kbps	kilobits per second	Ω	ohms
kHz	kilohertz	pF	picofarads
kΩ	kilohms	V	volts
M	1,048,576	VAC	volts alternating current
mA	milliamperes	VDC	volts direct current
MB	megabytes	W	watts

Other abbreviations used in this note include

$*n*$	hexadecimal value *n*
AC	alternating current
ADB	Apple Desktop Bus
API	application program interface
ASIC	application-specific integrated circuit
ATA	AT attachment
ATAPI	ATA packet interface
AUI	auxiliary unit interface
BCD	binary coded decimal
CAS	column address strobe (a memory control signal)
CCFL	cold cathode fluorescent lamp
CD	compact disc
CIS	card information structure
CLUT	color lookup table
CMOS	complementary metal oxide semiconductor
CPU	central processing unit
CSC	color screen controller
DAA	data access adapter (a telephone line interface)
DAC	digital-to-analog converter
DC	direct current
DCE	device control entry (a data structure)
DDM	driver descriptor map
DOS	disk operating system
DRAM	dynamic RAM
DSP	digital signal processor
FIFO	first in, first out
FPU	floating-point unit
FSTN	film supertwist nematic (a type of LCD)
HBA	host bus adapter
IC	integrated circuit
IDE	integrated device electronics
I/O	input/output
IR	infrared
LCD	liquid crystal display
LS TTL	low-power Schottky TTL (a standard type of device)
MMU	memory management unit

continued

NiCad	nickel cadmium
NiMH	nickel metal hydride
PCMCIA	Personal Computer Memory Card International Association
PDS	processor-direct slot
PROM	programmable read-only memory
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)
RISC	reduced instruction set computing
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
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

C H A P T E R 1

Introduction

The Macintosh PowerBook 5300 computer is the first of a new generation of all-in-one notebook computers featuring the PowerPC™ 603 microprocessor. Inside the computer's contoured case are a PCMCIA slot, an expansion bay for a floppy disk drive or other device, and space for a rechargeable battery.

Features

Here is a summary of the major features of the Macintosh PowerBook 5300 computer. Each feature is described more fully later in this developer note.

- **Processor**: The computer has a PowerPC 603 microprocessor running at a clock frequency of 100 or 117 MHz, depending on the model.
- **RAM:** The built-in memory consists of 8, 16, or 32 MB of low-power, self-refreshing dynamic RAM (DRAM).
- **RAM expansion:** The computer accepts a RAM expansion card with up to 56 MB, for a total of 64 MB of RAM.
- **Display:** The computer has a built-in flat panel display, an LCD backlit by a cold cathode fluorescent lamp (CCFL). The display can be one of three types: active-matrix color, DualScan color, or supertwist grayscale.
- **Video output:** The computer has built-in video output circuitry that provides a 256-color display on all Apple monitors up to 17 inches in size.
- **Hard disk:** The computer has one internal 2.5-inch IDE hard disk drive with a storage capacity of 500 MB to 1.1 GB. See "Peripheral Devices" on page 4.
- **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 Apple SuperDrive, some other IDE device, or a power device such as an AC adapter.
- **PCMCIA slot:** The computer accepts one type III or two type II PCMCIA cards.
- **Modem:** The computer accepts a PCMCIA modem card.
- **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, stereo audio input and output jacks, and a video output connector.
- **Networking:** The computer has a built-in LocalTalk network interface.
- **Sound:** The computer has a built-in microphone and speaker as well as a line-level input jack and 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.
- **Infrared link:** The computer has an infrared module that can communicate with Newton PDAs and other communications devices.
- **Batteries:** The computer has space for one Macintosh PowerBook Intelligent Battery. The battery is a 16.8-V lithium ion rechargeable battery with a built-in processor that communicates with the computer's Power Manager.
- **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 5300 computer has a streamlined case that opens up like a clamshell. Figure 1-1 shows a front view and Figure 1-2 shows a back view.

Figure 1-1 Front view of the computer

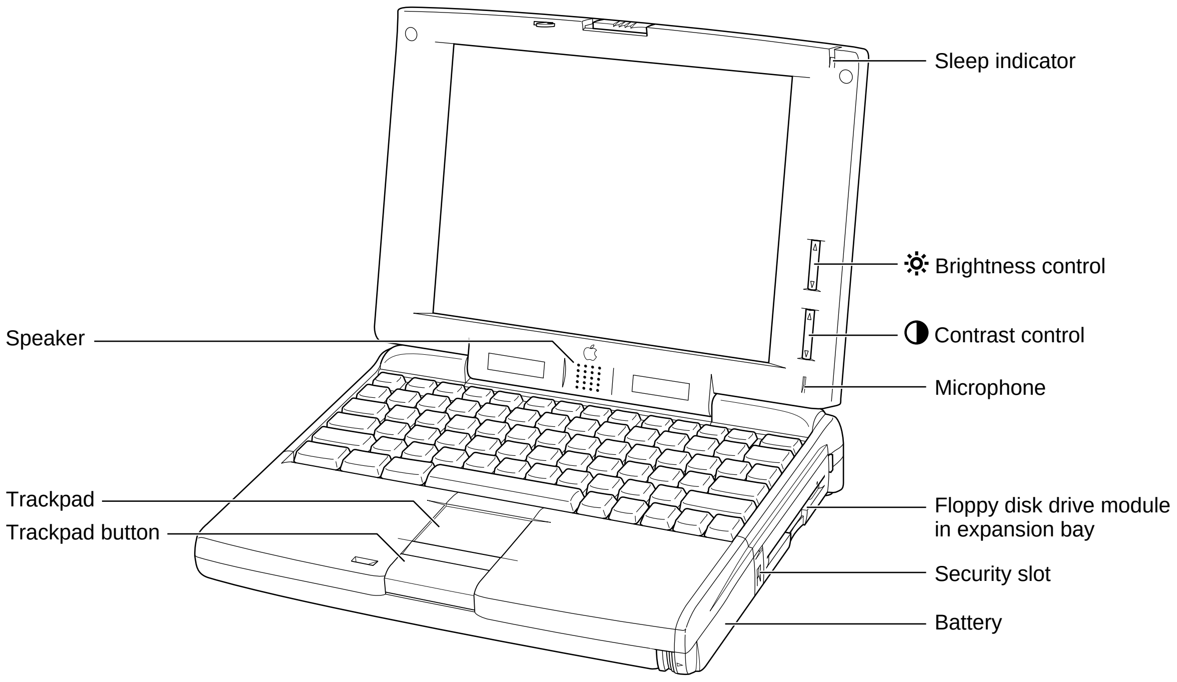
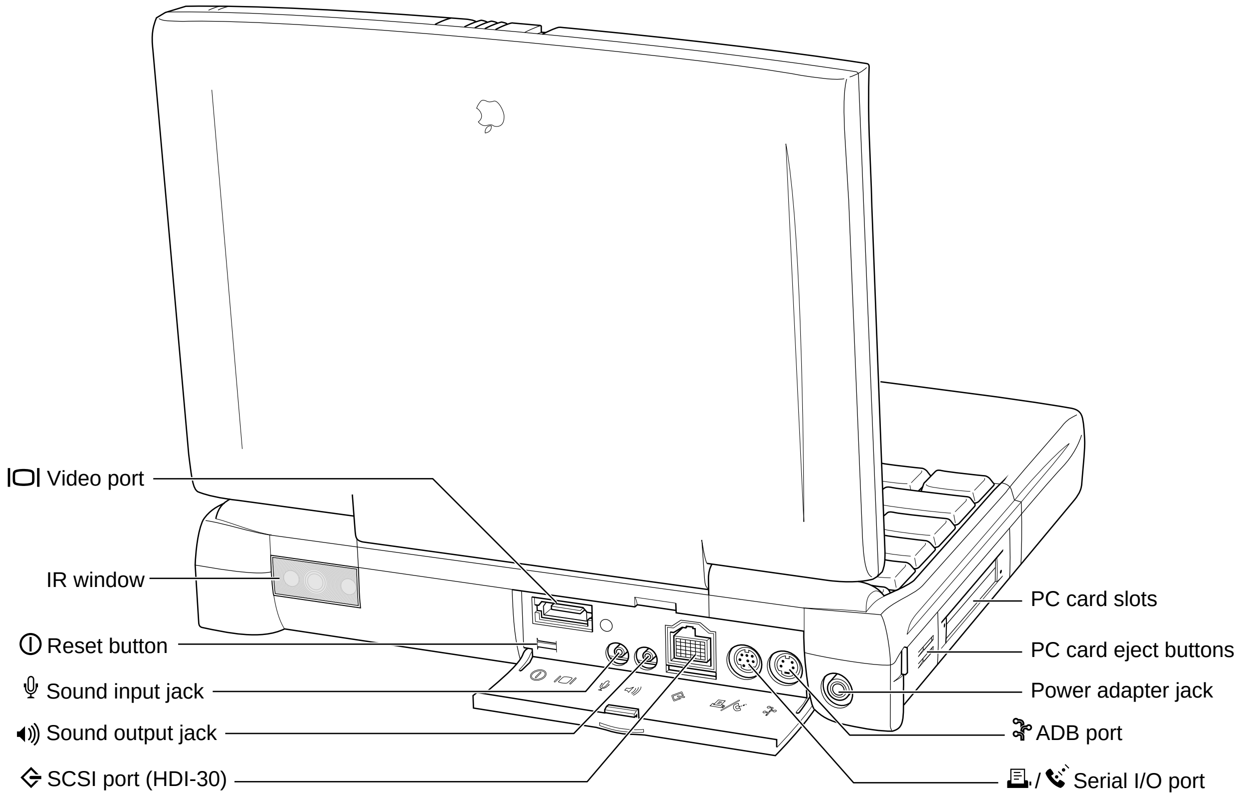


Figure 1-2 Back view of the computer



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 in the Macintosh PowerBook 5300 computer to 16 or 24 MB.

Note
In the 32-MB models, the RAM expansion slot is already occupied. ◆

- The Macintosh PowerBook Intelligent Battery is 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 two internal batteries in just four hours while the computer is running or two hours while the computer is shut down or in sleep mode.

Configurations

The Macintosh PowerBook 5300 computer is available in several configurations, as shown in Table 1-1.

Table 1-1 Configurations

Model number	Clock speed	RAM size	Hard disk size	Display size (pixels)	Display type
5300/100	100 MHz	8 MB	500 MB	640 by 480	DualScan gray scale
5300cs/100	100 MHz	8 MB	500 MB	640 by 480	DualScan color
5300cs/100	100 MHz	16 MB	750 MB	640 by 480	DualScan color
5300c/100	100 MHz	8 MB	500 MB	640 by 480	Active matrix color
5300c/100	100 MHz	16 MB	750 MB	640 by 480	Active matrix color
5300ce/117	117 MHz	32 MB	1.1 GB	800 by 600	Active matrix color

Compatibility Issues

The Macintosh PowerBook 5300 computer incorporates many significant changes from earlier PowerBook designs. This section highlights key areas you should investigate in order to ensure that your hardware and software work properly with the new PowerBook models. These topics are covered in more detail in subsequent sections.

RAM Expansion Cards

The RAM expansion card used in the Macintosh PowerBook 5300 computer is a new design. RAM expansion cards designed for earlier PowerBook models will not work in the PowerBook 5300 computer. 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 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 Display” 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 50 for more information about video modes.

Sound Sample Rates

The Macintosh PowerBook 5300 computer provides sound sample rates of 11.025 kHz, 22.05 kHz, and 44.1 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 5300 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*.

Developers should not assume that the Power Manager's data structures are the same on all 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

Microprocessor Differences

Differences between the PowerPC 603e and the PowerPC 601 microprocessor affect the way code is executed. Because of those differences, programs that execute correctly on the PowerPC 601 may cause problems on the PowerPC 603e.

Completion Serialized Instructions

Completion serialized instructions cannot be executed until all prior instructions have completed. The completion serialized instructions include load-and-store string and load-and-store multiple instructions. Such instructions can cause performance degradation on the more heavily pipelined implementations.

Representatives of Apple Computer are working with compiler developers to establish guidelines for the appropriate use of these instructions.

Split Cache

Unlike the PowerPC 601, which has a unified cache, the PowerPC 603e has separate caches for instructions and data. This can lead to cache coherency problems in applications that mix code and data.

In the Mac OS, almost all native code is loaded by the Code Fragment Manager, which ensures that the code is suitable for execution. If all your code is loaded by the Code Fragment Manager, you don't have to worry about cache coherency.

Problems can arise in applications that generate code in memory for execution. Examples include compilers that generate code for immediate execution and interpreters that translate code in memory for execution. For such cases, you can notify the Mac OS that data is subject to execution by using the call `MakeDataExecutable`, which is defined in OSUtils.h.

Data Alignment

In PowerPC systems, data is normally aligned on 32-bit boundaries, whereas data for the 680x0 is typically aligned on 16-bit boundaries. Even though the PowerPC was designed to support the 680x0 type of data alignment, misaligned data causes some performance degradation. Furthermore, performance with misaligned data varies across the different implementations of the PowerPC.

While it is essential to use 16-bit alignment for data that is being shared with 680x0 code, you should use PowerPC alignment for all other kinds of data. In particular, you should not use global 680x0 alignment when compiling your PowerPC applications; instead, use alignment pragmas to turn on 680x0 alignment only when necessary.

POWER-Clean Code

Several POWER instructions were included in the instruction set of the PowerPC 601 as part of the transition from POWER to PowerPC. Those instructions are not included in the instructions set of the PowerPC 603e.

Compilers designed for the POWER instruction set have also been used to compile programs for the PowerPC. Most of those compilers have the option to suppress the generation of the offending instructions. For example, the IBM xlc C compiler and the xlC C++ compiler have the option `-qarch=ppc.` Developers who use those compilers must verify that the option is in effect for all pieces of code that is intended to run on the PowerPC 603e.

The system software traps POWER instructions and emulates them in software. While this POWER emulation keeps the system from crashing when it encounters a POWER instruction, performance suffers because of the emulation. Developers should ensure that their code is free of POWER instructions.

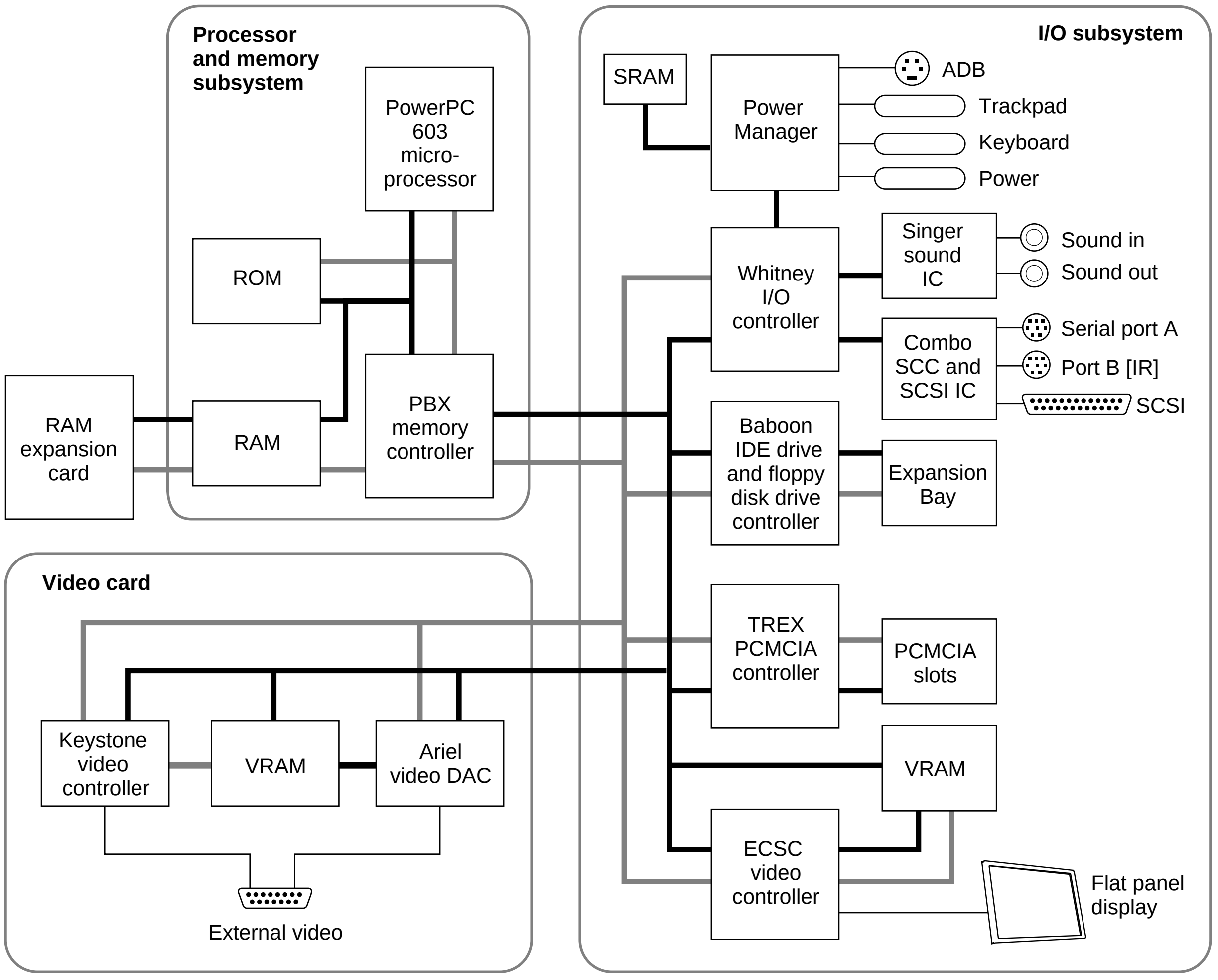
C H A P T E R 2

Architecture

The architecture of the Macintosh PowerBook 5300 computer is partitioned into three subsystems: the processor/memory subsystem, the input/output subsystem, and the video card. The processor/memory subsystem operates at 33.33 MHz on the PowerPC 603 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 PBX 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 in Figure 2-1 shows the subsystems and the modules that comprise them.

Figure 2-1 Block diagram



Processor/Memory Subsystem

The processor/memory subsystem includes the PowerPC 603 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.

Main Processor

The main processor in the Macintosh PowerBook 5300 computer is a PowerPC 603e microprocessor, an enhanced version of the PowerPC 603. Its principal features include

- full RISC processing architecture
- a load-store unit that operates in parallel with the processing units
- a branch manager that can usually implement branches by reloading the incoming instruction queue without using any processing time
- two internal memory management units (MMU), one for instructions and one for data
- two 16 KB on-chip caches for data and instructions

For complete technical details, see *Power PC 603 Microprocessor Implementation Definition Book IV.*

The PowerPC 603e microprocessor in the Macintosh PowerBook 5300 computer runs at a clock speed of either 100.00 or 116.66 (117) MHz, depending on the model. The micro-processor's clock speed is locked at either 3.0 or 3.5 times the memory subsystem's clock speed, which is 33.33 MHz.

RAM

The built-in RAM consists of 8, 16, or 32 MB of dynamic RAM (DRAM). The RAM ICs are 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 64 MB of RAM. The RAM expansion card for the Macintosh PowerBook 5300 computer is not compatible with the RAM card used in earlier PowerBook models. See the section "RAM Expansion" beginning on page 39 for details.

The PBX custom IC contains bank base registers that are used to make RAM addresses contiguous, starting at address $0000 0000. See "PBX Memory Controller IC" on page 12.

ROM

The ROM in the Macintosh PowerBook 5300 computer is implemented as a 1M by 32-bit array consisting of two 1 M by 16-bit ROM ICs. The ROM devices support burst mode so they do not degrade the performance of the PowerPC 603 microprocessor. The ROM ICs provide 4 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.

PBX Memory Controller IC

The PBX IC is a new Apple custom IC that provides RAM and ROM memory control and also acts as the bridge between the processor bus on the processor and memory subsystem and the 68030-type I/O bus on the main logic board. The PBX IC also provides bus cycle decoding for the SWIM floppy-disk controller.

Memory Control

The PBX IC controls the system RAM and ROM and provides address multiplexing and refresh signals for the DRAM devices. For information about the address multiplexing, see "Address Multiplexing" on page 43.

The PBX IC has a memory bank decoder in the form of an indexed register file. Each nibble in the register file represents a 2 MB page in the memory address space (64 MB). The value in each nibble maps the corresponding page to one of the eight banks of physical RAM. By writing the appropriate values into the register file at startup time, the system software makes the memory addresses contiguous.

Bus Bridge

The PBX 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 PBX 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 33.33 MHz and the I/O bus operates at 25.00 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 Combo I/O controller IC
- the Singer sound controller IC
- the Power Manager IC
- the display controller IC (ECSC)
- the Baboon custom IC that controls the expansion bay
- the TREX custom IC that controls the PCMCIA slots

The next 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

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 sent in or out through the Singer IC. For information about the operation of the Singer IC, see the section “Sound System” on page 28.

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:

- controlling sleep, shutdown, and on/off modes
- controlling power to the other ICs
- controlling clock signals to the other ICs
- supporting the ADB
- scanning the keyboard
- controlling display brightness
- monitoring battery charge level
- controlling battery charging

Display Controller IC

An ECSC (enhanced color support chip) IC provides the data and control interface to the LCD panel. The ECSC IC is similar to the CSC used in the PowerBook 520 and 540 models except that it can address 1 MB of video RAM. The ECSC 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 internal IDE drive see the section "Internal IDE 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," particularly Table 4-2 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," particularly Table 4-2 on page 36.

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 58.

Video Card

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

- the Ariel custom video controller IC
- the Keystone 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. 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 49.

C H A P T E R 3

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 5300 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 IDE Hard Disk Drive

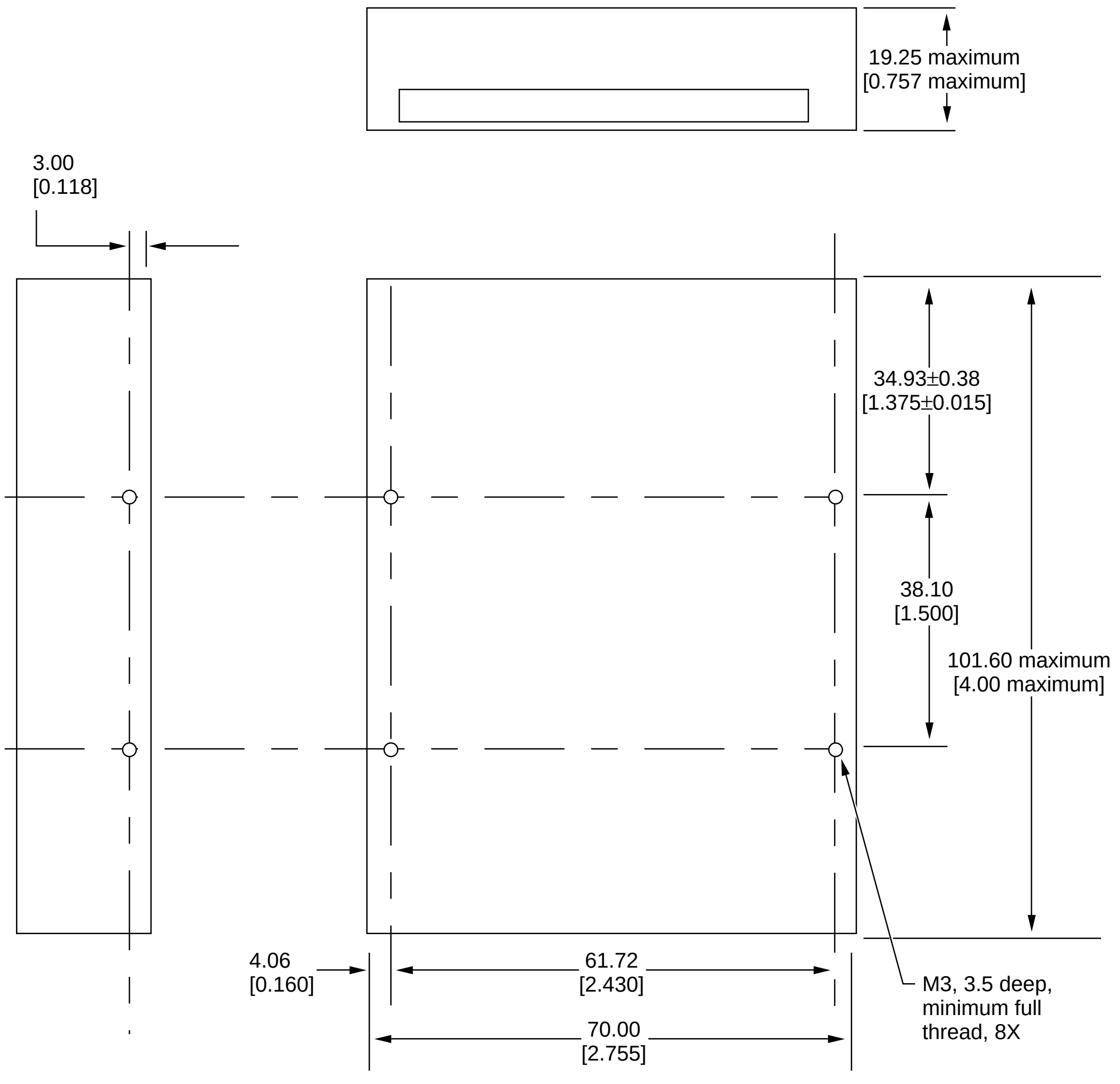
The Macintosh PowerBook 5300 computer has an internal hard disk that uses the standard IDE (integrated drive electronics) interface. This interface, used for IDE drives on IBM AT–compatible computers, is also referred to as the ATA interface. The implementation of the ATA interface on the Macintosh PowerBook 5300 computer is a subset of the ATA/IDE specification, ANSI proposal X3T10/0948D, Revision 2K (ATA-2).

For information about the IDE software interface, see Chapter 8, "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 IDE hard disk

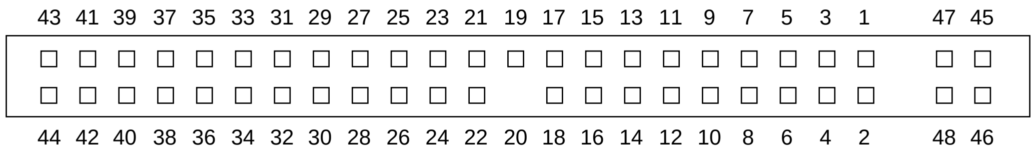


Note: Dimensions are in millimeters [inches]

Hard Disk Connector

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

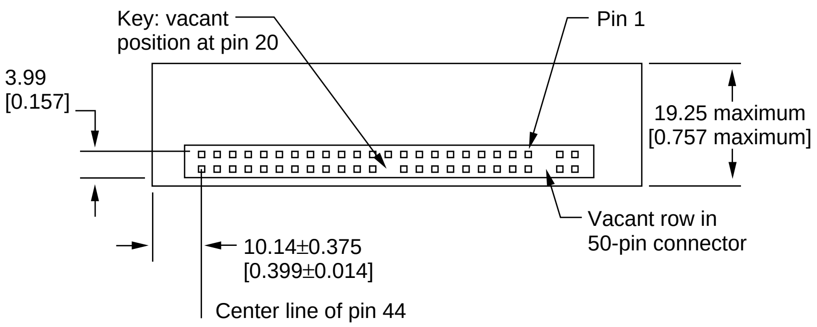


Note: gaps are equivalent to missing pins.

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


Key: vacant position at pin 20
Pin 1
3.99 [0.157]
19.25 maximum [0.757 maximum]
Vacant row in 50-pin connector
10.14±0.375 [0.399±0.014]
Center line of pin 44

Note: Dimensions are in millimeters [inches]

Signal Assignments

Table 3-1 shows the signal assignments on the 44-pin portion 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 IDE 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

continued

Table 3-1 Pin assignments on the IDE hard disk connector (continued)

Pin number	Signal name	Pin number	Signal name
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

IDE Signal Descriptions

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

Table 3-2 Signals on the IDE hard disk connector

Signal name	Signal description
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.

continued

Table 3-2 Signals on the IDE hard disk connector (continued)

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.
/DASP	Device active or slave present.
IORDY	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.
/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.
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.
/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.
Key	This pin is the key for the connector.

The IDE data bus is connected to the I/O bus through bidirectional bus buffers. To match the big-endian format of the 68030-compatible I/O bus, the bytes are swapped. The lowest 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 highest 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).

Trackpad

The pointing device in the Macintosh PowerBook 5300 computer is a trackpad, an integrated flat pad that replaces the trackball used in previous PowerBook computers. The trackpad 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 PowerBook computers, and it consumes less power.

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

Keyboard

A new keyboard design provides 76 (United States) or 77 (ISO) 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, United States layout

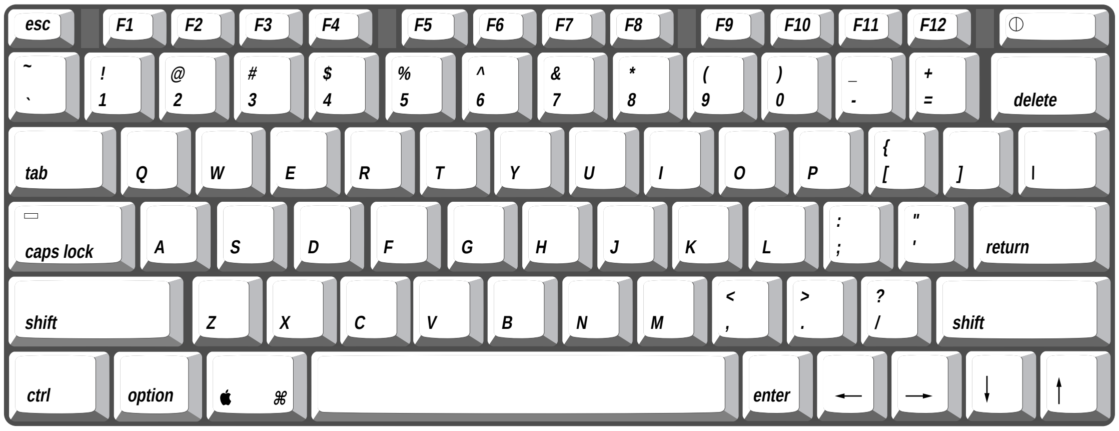
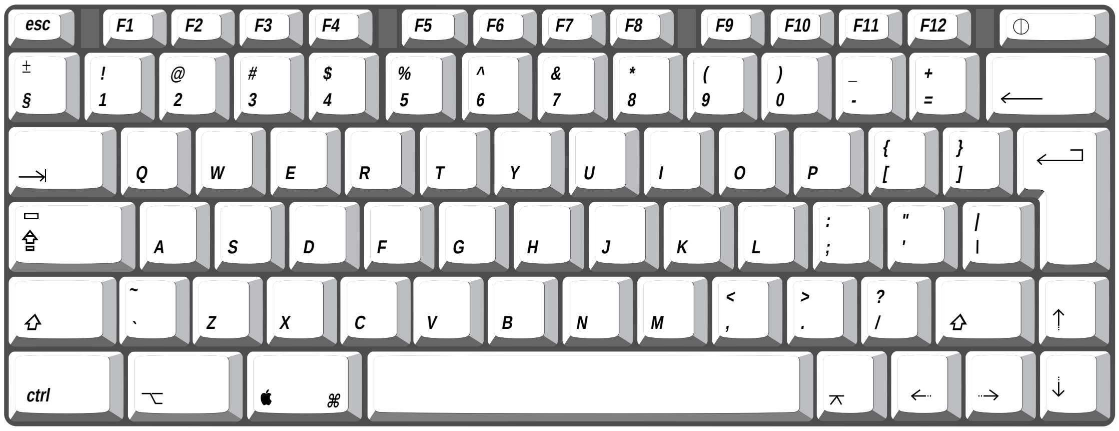

esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + = delete
tab Q W E R T Y U I O P { [ ] \
caps lock A S D F G H J K L : ; " ' return
shift Z X C V B N M < , > . ? / shift
ctrl option ⌘ enter

Figure 3-5 Keyboard, ISO layout


esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12
± § ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + =
Q W E R T Y U I O P { [}]
A S D F G H J K L : ; " ' | \
~ ` Z X C V B N M < , > . ? /
ctrl ⌘

After removing two screws, the user can lift out the keyboard to obtain access to the internal components and expansion connectors inside the Macintosh PowerBook 5300 computer.

Flat Panel Display

The Macintosh PowerBook 5300 computer has a built-in flat panel display. Four display options are available, as shown in Table 3-3. All four displays are backlit by a cold cathode fluorescent lamp (CCFL). The FSTN displays can show up to 256 colors on color displays or 16 levels of gray on grayscale displays. Both active matrix displays can show up to thousands of colors.

Table 3-3 Characteristics of the displays

Display type	Size (inches)	Size (pixels)	Dot pitch (mm)	Number of colors or grays
Supertwist grayscale (FSTN)	9.5	640 x 480	0.27	16
DualScan color (FSTN)	10.4	640 x 480	0.30	256
Active matrix color (TFT)	10.4	640 x 480	0.30	Thousands
Active matrix color (TFT)	10.4	800 x 600	0.27	Thousands

Flat Panel Display Circuitry

The flat panel display circuitry in the Macintosh PowerBook 5300 computer 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 new ECSC controller IC and a display buffer consisting of 1 MB of VRAM. The LCD display is compatible with software that uses QuickDraw and the Palette Manager. The display supports color table animation.

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.

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.

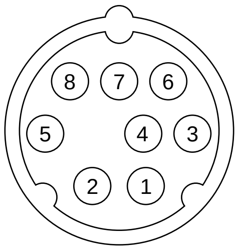
Passive matrix refers to a display technology that does not have individual transistors. That technology is also called FSTN, for film supertwist nematic, sometimes shortened to just supertwist.

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

Serial Port

The Macintosh PowerBook 5300 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 and Table 3-4 shows the signal assignments for the serial port.

Figure 3-6 Serial port connector



SCSI Port

The SCSI port on the Macintosh PowerBook 5300 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 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 3-4 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 +

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 SCSI connector signals

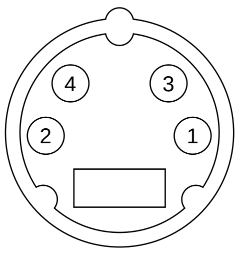
Pin number	SCSI connector	Pin number	SCSI connector
1	/SCSI.DISK.MODE	16	/DB6
2	/DB0	17	GND
3	GND	18	/DB7
4	/DB1	19	/DBP
5	TERMPWR (not used; reserved for future use)	20	GND
6	/DB2	21	/REQ
7	/DB3	22	GND
8	GND	23	/BSY
9	/ACK	24	GND
10	GND	25	/ATN
11	/DB4	26	/C/D
12	GND	27	/RST
13	GND	28	/MSG
14	/DB5	29	/SEL
15	GND	30	/I/O

ADB Port

The Apple Desktop Bus (ADB) port on the Macintosh PowerBook 5300 computer is functionally the same as on other Macintosh computers.

The ADB connector is a 4-pin mini-DIN connector. Figure 3-7 shows the arrangement of the pins on the ADB connector.

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 computer has an infrared (IR) module connected internally to serial port B. The IR module 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 Macintosh PowerBook 5300 computer, a Newton PDA, or some other IR-equipped device such as a remote control for a TV set.

The IR module in the Macintosh PowerBook 5300 computer supports the following communications protocols:

- LocalTalk
- Newton/Sharp/ASK
- HP/IRDA
- TV remote control (receive only)

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 5300 computer, the Singer codec supports two channels of digital sound with sample sizes up to 16 bits and sample rates of 11.025 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.

Note
All sound level specifications in this section are rms values. ◆

Sound Inputs

The sound system accepts inputs from several sources:

- built-in microphone
- external sound input jack
- 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 external sound input jack go through an analog multiplexer that selects either the external signals or the sound signals from the expansion bay. The multiplexer also lowers the levels of the maximum 2-V signal at the input jack to match the 1-V level of the codec circuits in the Singer IC.

The sound input jack has the following electrical characteristics:

- input impedance: 6.8k
- maximum level: 2.0 V rms

Note
The sound input jack accepts the maximum sound output of an audio CD without clipping. When working with sound sources that have significantly lower levels, you may wish to increase the sound output level. You can do that using the Sound Manager as described in *Inside Macintosh: Sound.* ◆

Stereo sound signals from the expansion bay go through an analog multiplexer that selects either those signals or the line signals from the external input jack. The multiplexer also 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
- maximum level: 0.5 V rms

Each PCMCIA card has one sound output pin (SPKR_OUT) and the computer accepts either one or two cards. The one-bit digital signals from the sound output pins are exclusive-ORed together and routed to the built-in speaker and the external sound output jack.

Sound Outputs

The sound system sends computer-generated sounds or sounds from an expansion-bay device 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.

C H A P T E R 4

Expansion Modules

This chapter describes each of the following expansion features of the Macintosh PowerBook 5300 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 5300 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

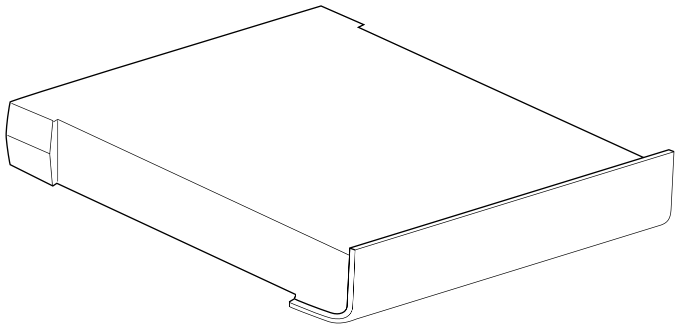
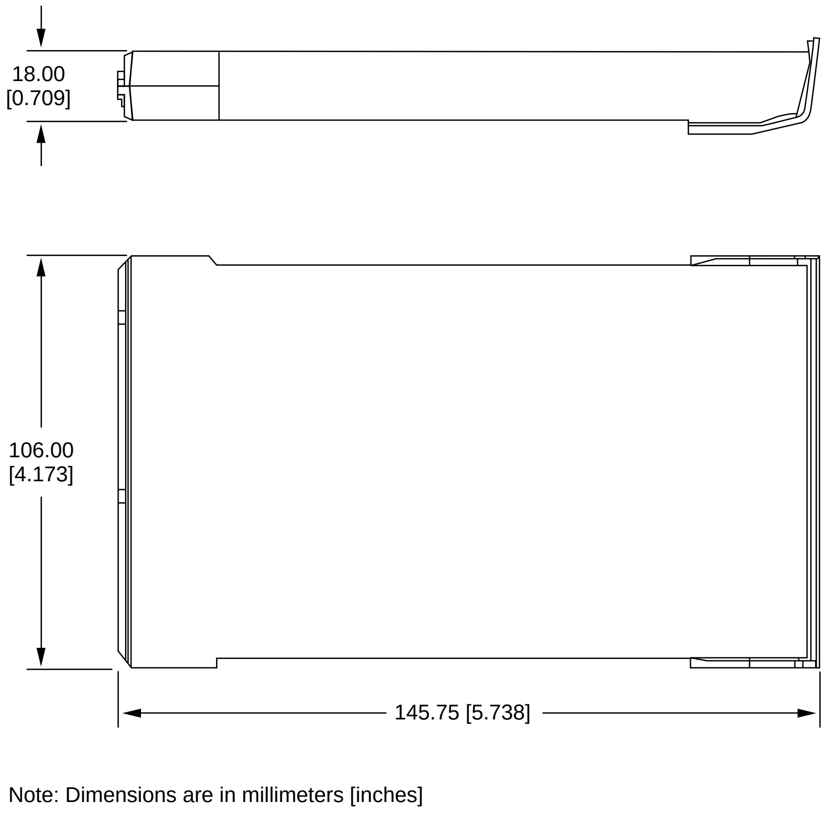


Figure 4-2 Expansion bay dimensions



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 first connection is the ground by way of the connector shells, then the power pins make contact, and last of all the signal lines. ▲

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

continued

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

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

Table 4-2 Control signals on the expansion bay connector

Signal name	Signal description
DEV_ID(2:0)	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(3:0)	Phase(2:0) 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.

continued

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

Signal name	Signal description
/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.
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.

Note
Signal names that begin with a slash (/) are active low. ◆

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 the /MB_PWR_EN signal low, which 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 the /MB_OE 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 set /MB_OE high, disabling the IDE bus.
2. System software responds to the interrupt by reading the device ID settings in the Power Manager IC, setting the /MB_PWR_EN signal high to turn off the power to the expansion bay, and asserting the /MB_IDE_RST 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 5300 computer.

The RAM expansion card can contain from 8 MB to 48 MB of self-refreshing dynamic RAM in one to six 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 5300 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 5300 computer. The mechanical specifications are given in a subsequent section, beginning on page 47.

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.

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)

continued

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

Pin	Signal name	Pin	Signal name
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)
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)

continued

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

Pin	Signal name	Pin	Signal name
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

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 ± 5%; 500 mA maximum.
+3V_MAIN	3.6 V ± 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.)

continued

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

Signal name	Description
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.

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 5300 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 100-Ω 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 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 address-bit numbers for those unused bits in italics; bit numbers for the bits that are used are shown in bold. ▲

Table 4-10 Address multiplexing for some typical DRAM devices

	Individual signals on DRAM_ADDR bus											
Type of DRAM device	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
4 M by 1 or 4 M by 4;11 row bits, 11 column bits												
Row address bits	*21*	**22**	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	**23**	**21**	**19**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
Device address pins	—	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
2 M by 8; 12 row bits, 9 column bits												
Row address bits	**21**	**22**	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	*23*	*21*	**19**	**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 by 8; 11 row bits, 10 column bits												
Row address bits	*21*	**22**	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	*23*	**21**	**19**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
Device address pins	—	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1 M by 4 or 1 M by 16; 11 row bits, 9 column bits												
Row address bits	**21**	*22*	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	*23*	*21*	**19**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
Device address pins	**10**	—	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1 M by 4 or 1 M by 16; 10 row bits, 10 column bits												
Row address bits	*21*	*22*	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	*23*	**21**	**19**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
Device address pins	—	—	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

continued

Table 4-10 Address multiplexing for some typical DRAM devices (continued)

	Individual signals on DRAM_ADDR bus											
Type of DRAM device	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
512K by 8; 10 row bits, 9 column bits												
Row address bits	*21*	*22*	**20**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**
Column address bits	*19*	*23*	*21*	**19**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**
Device address pins	**—**	**—**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Note
The address multiplexing scheme used in the Macintosh PowerBook 5300 computer supports only the types of RAM devices shown in Table 4-10. Other RAM types should not be used. ◆

Banks of DRAM

The DRAM expansion card can have up to six banks of RAM, selected by individual signals /RASL(1–3) and /RASH(1–3). 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.

Note
The PBX IC has a memory bank decoder that is used by the startup software to make the memory banks contiguous. For more information, see "Memory Control" on page 12. ◆

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.

Note
The DRAM refresh operation depends on the state of the computer. When the computer is operating normally, the PBX IC provides refresh signals consisting of 2048 CAS before RAS cycles every 128 ms. When the computer goes into sleep mode, the PBX IC switches the DRAM devices to their self-refresh feature to save power. See also “PBX Memory Controller IC” on page 12. ◆

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

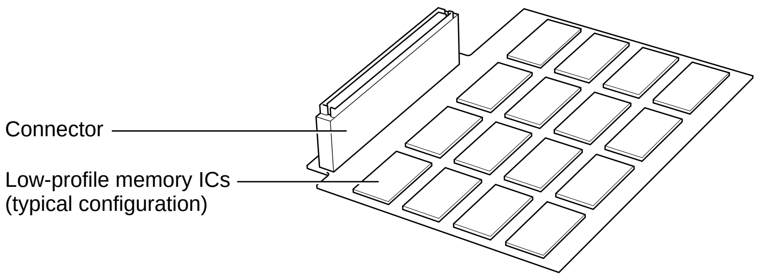
If the total capacitive loading for the devices on your card exceeds these guidelines, you should use buffers (such as 244-type devices) on the address and /RAS lines. Because of timing constraints, you cannot use buffers on the /CAS and /WE lines. If you do use buffers, you must keep within the following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

- Maximum delay on RA(): 8ns
- Maximum delay on /RASL() and /RASH(): 10ns
- 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



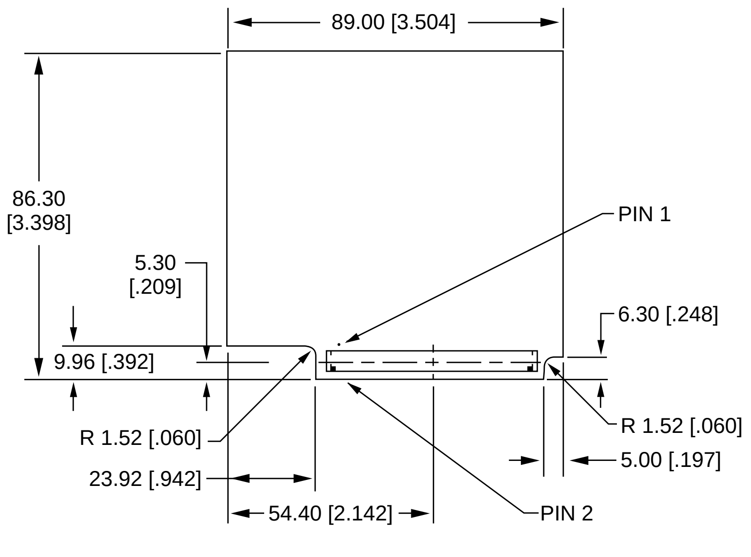
IMPORTANT

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.

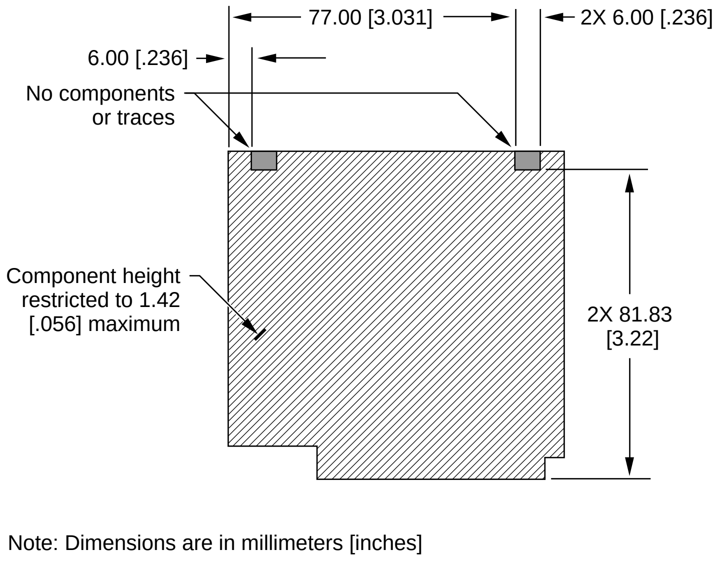
Figure 4-4 Dimensions of the RAM expansion card



Note: Dimensions are in millimeters [inches]

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


77.00 [3.031]
2X 6.00 [.236]
6.00 [.236]
No components or traces
Component height restricted to 1.42 [.056] maximum
2X 81.83 [3.22]
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.

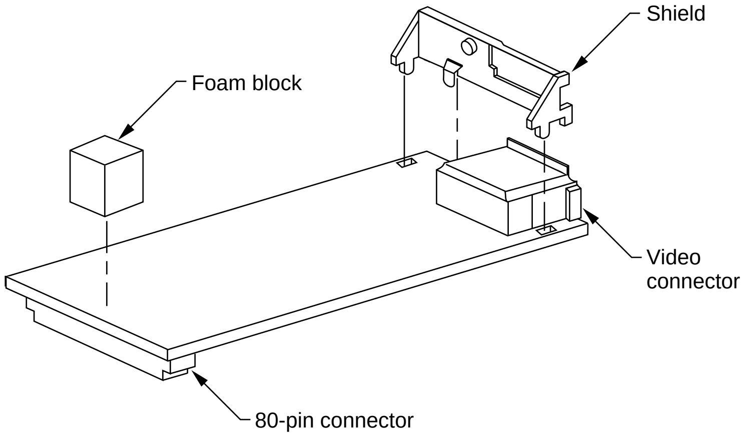
Video Card

The Macintosh PowerBook 5300 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 5300 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.

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 5300 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.

External Video Connector

The video card for the Macintosh PowerBook 5300 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

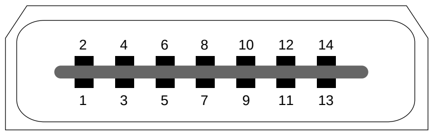
Pin			
VID-14	DB-15	**Signal name**	**Description**
1	2	RED.VID	Red video signal
2	1	RED.GND	Red video ground
3	4	SENSE0	Monitor sense signal 0
4	12	/VSYNC	Vertical synchronization signal
5	3	/CSYNC	Composite synchronization signal
6	11	GND	CSYNC and VSYNC ground
7	6	GRN.GND	Green video ground
8	5	GRN.VID	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

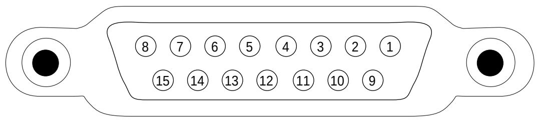
Hosiden America Corp.
10090 Pasadena Ave., Suite B2
Cupertino, CA 95014
408-252-0541

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 SENSE0-2 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

	Standard sense codes	**Extended sense codes**		
Monitor type	(2–0)	(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 5300 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

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(0)
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.

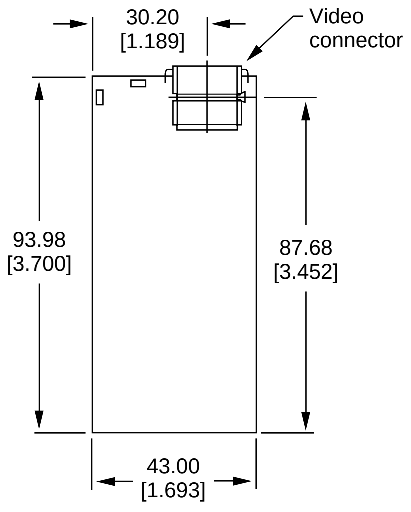
Table 4-15 Descriptions of the signals on the video card connector

Signal name	Description
/AS	Address strobe (68030 bus)
BUF_IOCLK	25 MHz I/O clock
/DSACK(1:0)	Bus data acknowledge (68030 bus)
/EXT_VID_CS	/CS for locations $FDXX XXXX
IO_ADDR(23:0)	Address bus (68030 bus)
IO_DATA(31:0)	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.

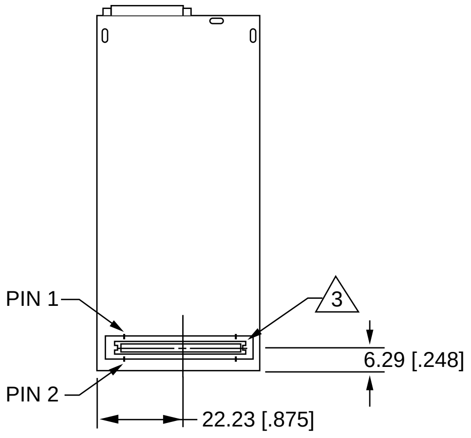
Figure 4-8 Dimensions of the video card



Note: Dimensions are in millimeters [inches]

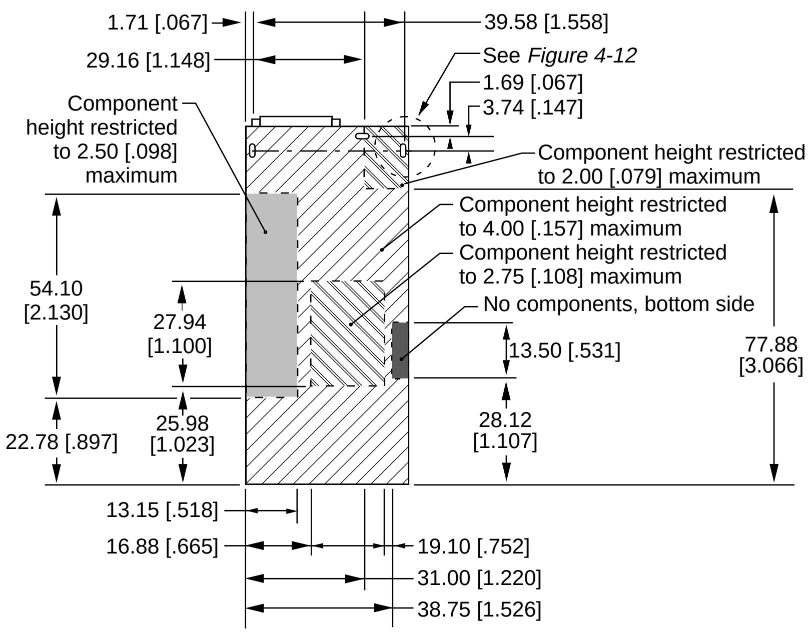
Figure 4-9 is a bottom view of the video card and shows the position of the 80-pin connector (callout 3). Figure 4-10 and Figure 4-11 show the component restrictions on the bottom and top of the card.

Figure 4-9 Video card and 80-pin connector



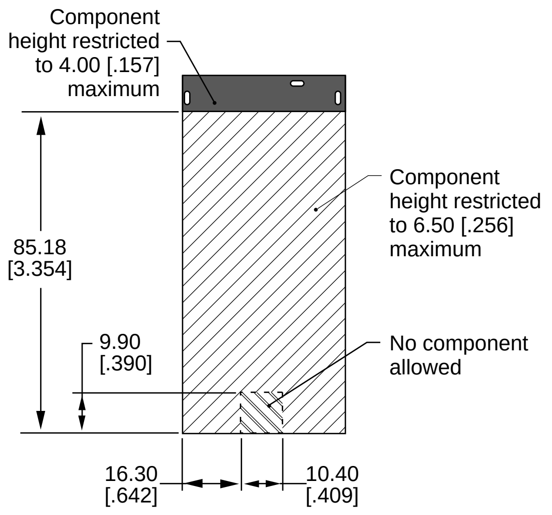
Note: Dimensions are in millimeters [inches]

Figure 4-10 Video card bottom view with component restrictions



Note: Dimensions are in millimeters [inches]

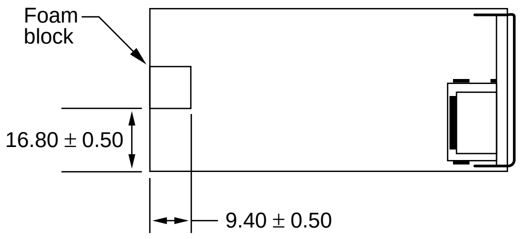
Figure 4-11 Video card top view with component restrictions


Component height restricted to 4.00 [.157] maximum
Component height restricted to 6.50 [.256] maximum
85.18 [3.354]
9.90 [.390]
No component allowed
16.30 [.642]
10.40 [.409]

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.

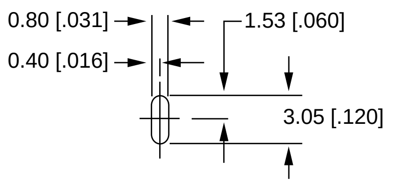
Figure 4-12 Video card top view



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 5300 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 9, "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 clicking on the Eject option in a Finder menu or by dragging the card's icon to the trash.

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 5300 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 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 µS)
- IRQ: Interrupt request, level mode only (pulse mode is not supported)
- SPKR: Speaker (digital audio output)
- STSCHG/RI: Status change and ring indicator (wake-up mode)
- INPACK: This signal is 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.

C H A P T E R 5

Software Features

Software Features

This chapter describes the new features of the software for the Macintosh PowerBook 5300 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 5300 computer is based on the ROM used in previous PowerBook computers, with enhancements to support the new features. Some of the features this ROM supports include the following:

- PowerPC 603 microprocessor
- 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.

PowerPC 603 Microprocessor

The PowerPC 603 microprocessor has power saving modes similar to the power cycling and sleep modes of earlier PowerBook models. The ROM has been modified to include the additional traps needed to control the power modes of the microprocessor.

The Macintosh PowerBook 5300 computer does not provide the economode reduced speed feature found on the Macintosh PowerBook 160 and 180 models.

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 5300 computer is 128 (hexadecimal $80). *Inside Macintosh: Overview* describes the Gestalt Manager and tells how to use the `gestaltMachineType` value to obtain the machine name string.

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 PowerBook models. The memory initialization and size code have been rewritten to deal with

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

Power Manager Software

Changes to the Power Manager software include

- power cycling and sleep mode for the PowerPC 603 microprocessor
- support for the new lithium ion batteries
- support for turning on and off power to the Ethernet interface

The Macintosh PowerBook 5300 computer uses a modified version of the public API for power management described in *Inside Macintosh: Devices.* See Chapter 7, "Power Manager Interface."

Display Controller Software

The Macintosh PowerBook 5300 computer has a new custom IC, the ECSC (enhanced 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 49.

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 the 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 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 5300 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 does support the commands used in the Apple SCSI device driver and the new Drive Setup utility.

Note
The ATA driver is described in Chapter 8, "Software for ATA Devices." ◆

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 5300 computer has a row of 12 function keys across the top. Except for the function keys, the keyboard is similar to those on previous 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 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 5300 computer is shipped with new system software based on Mac OS version 7.5 and augmented by several new features.

IMPORTANT
Even though the software for the Macintosh PowerBook 5300 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
- a new Dynamic Recompilation Emulator
- a Resource Manager completely in native code
- improved math library
- POWER-clean native code
- POWER emulation
- 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 5300 computer has the status and control element called the control strip that was introduced in the PowerBook 280 and the PowerBook 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 Hard Disk device 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 Hard Disk device driver and the ATA Manager are described in Chapter 8, “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 `XIOParam`).

For the definitions of the new API and the three data types, please see “The API Modifications” beginning on page 77.

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
- 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)
- Drive 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 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:

- 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 5300 computer includes a new disk setup utility named Drive Setup that replaces the old HDSC Setup utility. 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.

Dynamic Recompilation Emulator

The Dynamic Recompilation Emulator (or DR Emulator) is an extension to the current interpretive emulator providing on-the-fly translation of 680x0 instructions into PowerPC instructions for increased performance. The DR Emulator operates as an enhancement to a modified version of the existing interpretive emulator.

The design of the DR Emulator mimics a hardware instruction cache and employs a variable size translation cache. Each compiled 680x0 instruction requires on average fewer than four PowerPC instructions. In operation, the DR Emulator depends on locality of execution to make up for the extra cycles used in translating the code.

The DR Emulator provides a high degree of compatibility for 680x0 code. One area where compatibility will be less than that of the current interpretive emulator is for self-modifying code that does not call the cache flushing routines. Such code also has compatibility problems on Macintosh Quadra models with the cache enabled.

Resource Manager in Native Code

The Resource Manager in the software for the Macintosh PowerBook 5300 computer is similar to the one in the earlier Power Macintosh computers except that it is completely in native PowerPC code. Because the Resource Manager is used intensively by both system software and applications, the native version provides an improvement in system performance.

The Process Manager has been modified to remove patches it formerly made to the Resource Manager.

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 new math library has been improved to better exploit the floating-point features of the PowerPC microprocessor. The math library now includes enhancements that assist the compiler in carrying out its register allocation, branch prediction, and overlapping of integer and floating-point operations.

Compared with the previous version, the new math library provides much improved performance without compromising its accuracy or robustness. It provides performance gains for often-used functions of up to 15 times.

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

New BlockMove Extensions

The system software for the Macintosh PowerBook 5300 computer includes new extensions to the `BlockMove` routine. The extensions provide improved performance for programs running in native mode.

The new `BlockMove` extensions provide several benefits for developers.

- They're optimized for the PowerPC 603 and PowerPC 604 processors, rather than the PowerPC 601.
- They're compatible with the new Dynamic Recompilation Emulator.
- They provide a way to handle cache-inhibited address spaces.
- They include new high-speed routines for setting memory to zero.

Note
The new BlockMove extensions do not use the string instructions, which are fast on the PowerPC 601 but slow on other PowerPC implementations. ◆

Some of the new BlockMove extensions can be called only from native code; see Table 5-1.

Except for BlockZero and BlockZeroUncached, the new BlockMove extensions use the same parameters as BlockMove. Calls to BlockZero and BlockZeroUncached have only two parameters, a pointer and a length; refer to the header file (Memory.h).

Table 5-1 summarizes the BlockMove routines and according to three criteria: whether the routine can be called from 680x0 code, whether it is okay to use for moving 680x0 code, and whether it is okay to use with buffers or other uncacheable destination locations.

Table 5-1 Summary of BlockMove routines

BlockMove version	Can be called from 680x0 code	Okay to use for moving 680x0 code	Okay to use with buffers
BlockMove	Yes	Yes	No
BlockMoveData	Yes	No	No
BlockMoveDataUncached	No	No	Yes
BlockMoveUncached	No	Yes	Yes
BlockZero	No	—	No
BlockZeroUncached	No	—	Yes

The fastest way to move data is to use the BlockMoveData routine. It is the recommended method whenever you are certain that the data is cacheable and does not contain executable 680x0 code.

The BlockMove routine is slower than the BlockMoveData routine only because it has to clear out the software cache used by the DR Emulator. If the DR EMulator is not in use, the BlockMove routine and the BlockMoveData routine are the same.

IMPORTANT
The versions of BlockMove for cacheable data use the dcbz instruction to avoid unnecessary pre-fetch of destination cache blocks. For uncacheable data, you should avoid using those routines because the dcbz instruction faults and must be emulated on uncacheable or write-through locations, making execution extremely slow. ▲

IMPORTANT
Driver software cannot call the `BlockMove` routines directly. Instead, drivers must use the `BlockCopy` routine, which is part of the Driver Services Library. The `BlockCopy` routine is an abstraction that allows you to postpone binding the specific type of `BlockMove` operation until implementation time. ▲

The Driver Services Library is a collection of useful routines that Apple Computer provides for developers working with the new Power Macintosh models. For more information, please refer to *Designing PCI Cards and Drivers for Power Macintosh Computers.*

POWER-Clean Native Code

The instruction set of the PowerPC 601 microprocessor included some of the same instructions as those found in the instruction set of the POWER processor, and the compiler used to generate native code for the system software in the previous Power Macintosh models generated some of those POWER-only instructions. However, the PowerPC 603 microprocessor used in the Macintosh PowerBook 5300 computer does not support the POWER-only instructions, so a new POWER-clean version of the compiler is being used to compile the native code fragments.

Note
The term *POWER-clean* refers to code that is free of the POWER instructions that would prevent it from running correctly on a PowerPC 603 or PowerPC 604 microprocessor. ◆

Here is a list of the POWER-clean native code elements in the system software for the Macintosh PowerBook 5300 computer.

- interface library
- private interface library
- native QuickDraw
- MathLib
- Mixed Mode Manager
- Code Fragment Manager
- Font Dispatch
- Memory Manager
- standard text
- the `FMSwapFont` function
- Standard C Library

POWER Emulation

Earlier Power Macintosh computers included emulation for certain PowerPC 601 instructions that would otherwise cause an exception. The emulation code dealt with memory reference instructions to handle alignment and data storage exceptions. It also handled illegal instruction exceptions caused by some PowerPC instructions that were not implemented in the PowerPC 601. In the Macintosh PowerBook 5300 computer, the emulation code has been enhanced to include the POWER instructions that are implemented on the PowerPC 601 but not on the PowerPC 603.

Note

Although the term *POWER emulation* is often used, a more appropriate name for this feature is *PowerPC 601 compatibility.* Rather than supporting the entire POWER architecture, the goal is to support those features of the POWER architecture that are available to programs running in user mode on the PowerPC 601-based Power Macintosh computers. ◆

POWER-Clean Code

Because the emulation of the POWER-only instructions degrades performance, Apple Computer recommends that developers revise any applications that use those instructions to conform with the PowerPC architecture. POWER emulation works, but at a significant cost in performance; POWER-clean code is preferable.

Emulation and Exception Handling

When an exception occurs, the emulation code first checks to see whether the instruction encoding is supported by emulation. If it is not, the code passes the original cause of the exception (illegal instruction or privileged instruction) to the application as a native exception.

If the instruction is supported by emulation, the code then checks a flag bit to see whether emulation has been enabled. If emulation is not enabled at the time, the emulator generates an illegal instruction exception.

Code Fragments and Cache Coherency

Whereas the PowerPC 601 microprocessor has a single cache for both instructions and data, the PowerPC 603 has separate instruction and data caches. As long as applications deal with executable code by using the Code Fragment Manager, cache coherency is maintained. Applications that bypass the Code Fragment Manager and generate executable code in memory, and that do not use the proper cache synchronization instructions or Code Fragment Manager calls, are likely to encounter problems when running on the PowerPC 603.

IMPORTANT
The emulation software in the Macintosh PowerBook 5300 computer cannot make the separate caches in the PowerPC 603 behave like the combined cache in the PowerPC 601. Applications that generate executable code in memory must be modified to use the Code Fragment Manager or maintain proper cache synchronization by other means. ▲

Limitations of PowerPC 601 Compatibility

The emulation code in the Macintosh PowerBook 5300 computer allows programs compiled for the PowerPC 601 to execute without halting on an exception whenever they use a POWER-only feature. For most of those features, the emulation matches the results that are obtained on a Power Macintosh computer with a PowerPC 601. However, there are a few cases where the emulation is not an exact match; those cases are summarized here.

- **MQ register.** Emulation does not match the undefined state of this register after multiply and divide instructions.
- **`div` and `divo` instructions.** Emulation does not match undefined results after an overflow.
- **Real-time clock registers.** Emulation matches the 0.27 percent speed discrepancy of the Power Macintosh models that use the PowerPC 601 microprocessor, but the values of the low-order 7 bits are not 0.
- **POWER version of `dec` register.** Emulation includes the POWER version, but decrementing at a rate determined by the time base clock, not by the real-time clock.
- **Cache line compute size (`clcs`) instruction.** Emulation returns values appropriate for the type of PowerPC microprocessor.
- **Undefined SPR encodings.** Emulation does not ignore SPR encodings higher than 32.
- **Invalid forms.** Invalid combinations of register operands with certain instructions may produce results that do not match those of the PowerPC 601.
- **Floating-point status and control register (FPSCR).** The FPSCR in the PowerPC 601 does not fully conform to the PowerPC architecture, but the newer PowerPC processors do.

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.

C H A P T E R 6

Large Volume Support

Large Volume Support

This chapter describes the large volume file system for the Macintosh PowerBook 5300 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 that are greater 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 5300 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 77.

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.

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`).

```
struct XVolumeParam {
   ParamBlockHeader
   unsigned long    ioXVersion;      // XVolumeParam version == 0
   short            ioVolIndex;      // volume index
   unsigned long    ioVCrDate;       // date & time of creation
   unsigned long    ioVLsMod;        // date & 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

`ioVolIndex`	An index for use with the `PBHGetVInfo` function.
`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.
`ioVAlBlkSiz`	The size of allocation blocks.
`ioVClpSiz`	The clump size.
`ioAlBlSt`	The first block in the volume map.

`ioVNxtCNID`	The next unused catalog node ID.
`ioVFrBlk`	The number of unused allocation blocks.
`ioVSigWord`	A signature word identifying the type of volume; it's $D2D7 for MFS volumes and $4244 for volumes that support HFS calls.
`ioVDrvInfo`	The drive number of the drive containing the volume.
`ioVDRefNum`	For online volumes, the reference number of the I/O driver for the drive identified by `ioVDrvInfo`.
`ioVFSID`	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.
`ioVBkUp`	The date and time the volume was last backed up (it's 0 if never backed up).
`ioVSeqNum`	Used internally.
`ioVWrCnt`	The volume write count.
`ioVFilCnt`	The total number of files on the volume.
`ioVDirCnt`	The total number of directories (not including the root directory) on the volume.
`ioVFndrInfo`	Information used by the Finder.

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 in the parameter block is the parameter `ioWPosOffset`, which is of type `int64`.

```
struct XIOParam {
   QElemPtr      qLink;       // next queue entry
   short         qType;       // queue type
   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          ioPermssn;   // 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         ioWPosOffset;// wide positioning offset
};
```

Field descriptions

`ioRefNum`	The file reference number of an open file.
`ioVersNum`	A version number. This field is no longer used and you should always set it to 0.
`ioPermssn`	The access mode.
`ioMisc`	Depends 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 `ioMisc` is of type `Ptr`, you'll need to perform type coercion to interpret the value of `ioMisc` correctly when it contains an end-of-file (a `LongInt` value) or version number (a `SignedByte` value).
`ioBuffer`	A pointer to a data buffer into which data is written by `_Read` calls and from which data is read by `_Write` calls.
`ioReqCount`	The requested number of bytes to be read, written, or allocated.
`ioActCount`	The number of bytes actually read, written, or allocated.
`ioPosMode`	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 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 call, 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

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.
←	`ioVNmAlBlks`	`unsigned short`	Number of allocation blocks.
←	`ioVAlBlkSiz`	`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.
←	`ioVDRefNum`	`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.
←	`ioVFndrInfo[8]`	`long`	Used by the Finder.
←	`ioVTotalBytes`	`uint64`	Total number of bytes on the volume.
←	`ioVFreeBytes`	`uint64`	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
`_HFSDispatch`	$0012

RESULT CODES

`noErr`	0	Successful completion, no error occurred
`nsvErr`	–35	No such volume
`paramErr`	–50	No default volume

C H A P T E R 7

Power Manager Interface

Power Manager Interface

This chapter describes the new application programming interface (API) to the Power Manager control software in the Macintosh PowerBook 5300 computer.

About the Power Manager Interface

Developers have written control panel software for previous Macintosh PowerBook models to give the user more control over the power management settings than is provided in the PowerBook control panel. Because that software reads and writes directly to the Power Manager's private data structures and parameter RAM, the software needs to be updated any time Apple Computer makes a change to the internal operation of the Power Manager.

System software for the Macintosh PowerBook 5300 computer and for future Macintosh PowerBook models 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. The new routines provide access to most of the Power Manager's parameters. Some functions will be reserved because of their overall effect on the system. The interface is extensible; it will probably grow over time to acccommodate new kinds of functions.

Things That May Change

By using the Power Manager interface, developers can isolate themselves from future changes to the internal operation of the Power Manager software.

IMPORTANT
Apple Computer reserves the right to change the internal operation of the Power Manager software. Developers should not make their applications depend on the Power Manager's internal data structures or parameter RAM. ▲

As new PowerBook models appear, developers should not depend on the Power Manager's internal data structures staying the same. 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

Checking for Routines

Before calling any of the Power Manager interface routines, it's always a good idea to call the Gestalt Manager to see if they're present on the computer. The Gestalt Manager is described in *Inside Macintosh: Overview.*

A new bit has been added to the `gestaltPowerMgrAttr` selector:

```
#define gestaltPMgrDispatchExists 4
```

If that bit is set to 1, then the routines are present.

Because more routines may be added in the future, one of the new routines simply returns the number of routines that are implemented. The following code fragment determines both that the routines in general exist and that at least the hard disk spindown routine exists.

```
long   pmgrAttributes;
Boolean routinesExist;

routinesExist = false;
if (! Gestalt(gestaltPowerMgrAttr, &pmgrAttributes))
   if (pmgrAttributes & (1<<gestaltPMgrDispatchExists))
      if (PMSelectorCount() >= 7)
         routinesExist = true;
```

▲ **WARNING**
If you call a routine that does not exist, the call to the public Power Manager trap (if the trap exists) will return an error code, which your program could misinterpret as data. ▲

Power Manager Interface Routines

This section tells you how to call the interface routines for the Power Manager software. The interface routines are listed here in the order of their routine selector values, as shown in Table 7-1.

Table 7-1 Interface routines and their selector values

	Selector value	
Routine name	Decimal	Hexadecimal
`PMSelectorCount`	0	$00
`PMFeatures`	1	$01
`GetSleepTimeout`	2	$02
`SetSleepTimeout`	3	$03
`GetHardDiskTimeout`	4	$04
`SetHardDiskTimeout`	5	$05
`HardDiskPowered`	6	$06
`SpinDownHardDisk`	7	$07
`IsSpindownDisabled`	8	$08
`SetSpindownDisable`	9	$09
`HardDiskQInstall`	10	$0A
`HardDiskQRemove`	11	$0B
`GetScaledBatteryInfo`	12	$0C
`AutoSleepControl`	13	$0D
`GetIntModemInfo`	14	$0E
`SetIntModemState`	15	$0F
`MaximumProcessorSpeed`	16	$10
`CurrentProcessorSpeed`	17	$11
`FullProcessorSpeed`	18	$12
`SetProcessorSpeed`	19	$13
`GetSCSIDiskModeAddress`	20	$14
`SetSCSIDiskModeAddress`	21	$15
`GetWakeupTimer`	22	$16
`SetWakeupTimer`	23	$17
`IsProcessorCyclingEnabled`	24	$18
`EnableProcessorCycling`	25	$19
`BatteryCount`	26	$1A
`GetBatteryVoltage`	27	$1B
`GetBatteryTimes`	28	$1C

Assembly-language note

All the routines share a single trap, `_PowerMgrDispatch` ($A09E). The trap is register based; parameters are passed in register D0 and sometimes also in A0. A routine selector value passed in the low word of register D0 determines which routine is executed. ◆

PMSelectorCount

You can use the `PMSelectorCount` routine to determine which routines are implemented.

```
short PMSelectorCount();
```

DESCRIPTION

The `PMSelectorCount` routine returns the number of routine selectors present. Any routine whose selector value is greater than the returned value is not implemented.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `PMSelectorCount` is 0 ($00) in the low word of register D0. The number of selectors is returned in the low word of register D0.

PMFeatures

You can use the `PMFeatures` routine to find out which features of the Power Manager are implemented.

```
unsigned long PMFeatures();
```

DESCRIPTION

The `PMFeatures` routine returns a 32-bit field describing hardware and software features associated with the Power Manager on a particular machine. If a bit value is 1, that feature is supported or available; if the bit value is 0, that feature is not available. Unused bits are reserved by Apple for future expansion.

Field descriptions

Bit name	Bit number	Description
`hasWakeupTimer`	0	The wakeup timer is supported.
`hasSharedModemPort`	1	The hardware forces exclusive access to either SCC port A or the internal modem. (If this bit is not set, then typically port A and the internal modem may be used simultaneously by means of the Communications Toolbox.)
`hasProcessorCycling`	2	Processor cycling is supported; that is, when the computer is idle, the processor power is cycled to reduce the power usage.
`mustProcessorCycle`	3	The processor cycling feature must be left on (turn it off at your own risk).
`hasReducedSpeed`	4	Processor can be started up at a reduced speed in order to extend battery life.
`dynamicSpeedChange`	5	Processor speed can be switched dynamically between its full and reduced speed at any time, rather than only at startup time.
`hasSCSIDiskMode`	6	The SCSI disk mode is supported.
`canGetBatteryTime`	7	The computer can provide an estimate of the battery time remaining.
`canWakeupOnRing`	8	The computer supports waking up from the sleep state when an internal modem is installed and the modem detects a ring.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `PMFeatures` is 1 ($01) in the low word of register D0. The 32-bit field of supported features is returned in register D0.

GetSleepTimeout

You can use the `GetSleepTimeout` routine to find out how long the computer will wait before going to sleep.

```
unsigned char GetSleepTimeout();
```

DESCRIPTION

The `GetSleepTimeout` routine returns the amount of time that the computer will wait after the last user activity before going to sleep. The value of `GetSleepTimeout` is expressed as the number of 15-second intervals that the computer will wait before going to sleep.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetSleepTimeout` is 2 ($02) in the low word of register D0. The sleep timeout value is returned in the low word of register D0.

SetSleepTimeout

You can use the `SetSleepTimeout` routine to set how long the computer will wait before going to sleep.

```
void SetSleepTimeout(unsigned char timeout);
```

DESCRIPTION

The `SetSleepTimeout` routine sets the amount of time the computer will wait after the last user activity before going to sleep. The value of `SetSleepTimeout` is expressed as the number of 15-second intervals that make up the desired time. If a value of 0 is passed in, the routine sets the `timeout` value to the default value (currently equivalent to 8 minutes).

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSleepTimeout` is 3 ($03) in the low word of register D0. The sleep timeout value to set is passed in the high word of register D0.

GetHardDiskTimeout

You can use the `GetHardDiskTimeout` routine to find out how long the computer will wait before turning off power to the internal hard disk.

```
unsigned char GetHardDiskTimeout();
```

DESCRIPTION

The `GetHardDiskTimeout` routine returns the amount of time the computer will wait after the last use of a SCSI device before turning off power to the internal hard disk. The value of `GetHardDiskTimeout` is expressed as the number of 15-second intervals the computer will wait before turning off power to the internal hard disk.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetHardDiskTimeout` is 4 ($04) in the low word of register D0. The hard disk timeout value is returned in the low word of register D0.

SetHardDiskTimeout

You can use the `SetHardDiskTimeout` routine to set how long the computer will wait before turning off power to the internal hard disk.

```
void SetHardDiskTimeout(unsigned char timeout);
```

DESCRIPTION

The `SetHardDiskTimeout` routine sets how long the computer will wait after the last use of a SCSI device before turning off power to the internal hard disk. The value of `SetHardDiskTimeout` is expressed as the number of 15-second intervals the computer will wait before turning off power to the internal hard disk. If a value of 0 is passed in, the routine sets the `timeout` value to the default value (currently equivalent to 4 minutes).

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetHardDiskTimeout` is 5 ($05) in the low word of register D0. The hard disk timeout value to set is passed in the high word of register D0.

HardDiskPowered

You can use the `HardDiskPowered` routine to find out whether the internal hard disk is on.

```
Boolean HardDiskPowered();
```

DESCRIPTION

The `HardDiskPowered` routine returns a Boolean value indicating whether the internal hard disk is powered up. A value of `true` means that the hard disk is on, and a value of `false` means that the hard disk is off.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskPowered` is 6 ($06) in the low word of register D0. The Boolean result is returned in the low word of register D0.

SpinDownHardDisk

You can use the `SpinDownHardDisk` routine to force the hard disk to spin down.

```
void SpinDownHardDisk();
```

DESCRIPTION

The `SpinDownHardDisk` routine immediately forces the hard disk to spin down and power off if it was previously spinning. Calling `SpinDownHardDisk` will not spin down the hard disk if spindown is disabled by calling `SetSpindownDisable` (defined later in this section).

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SpinDownHardDisk` is 7 ($07) in the low word of register D0.

IsSpindownDisabled

You can use the `IsSpindownDisabled` routine to find out whether hard disk spindown is enabled.

```
Boolean IsSpindownDisabled();
```

DESCRIPTION

The `IsSpindownDisabled` routine returns a Boolean value of `true` if hard disk spindown is disabled, or `false` if spindown is enabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsSpindownDisabled` is 8 ($08) in the low word of register D0. The Boolean result is passed in the low byte of register D0.

SetSpindownDisable

You can use the `SetSpindownDisable` routine to disable hard disk spindown.

```
void SetSpindownDisable(Boolean setDisable);
```

DESCRIPTION

The `SetSpindownDisable` routine enables or disables hard disk spindown, depending on the value of `setDisable`. If the value of `setDisable` is `true`, hard disk spindown will be disabled; if the value is `false`, spindown will be enabled.

Disabling hard disk spindown affects the `SpinDownHardDisk` routine, defined earlier, as well as the normal spindown that occurs after a period of hard disk inactivity.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSpindownDisable` is 9 ($09) in the low word of register D0. The Boolean value to set is passed in the high word of register D0.

HardDiskQInstall

You can use the `HardDiskQInstall` routine to notify your software when power to the internal hard disk is about to be turned off.

```
OSErr HardDiskQInstall(HDQueueElement *theElement);
```

DESCRIPTION

The `HardDiskQInstall` routine installs an element into the hard disk power-down queue to provide notification to your software when the internal hard disk is about to be powered off. For example, this feature might be used by the driver for an external battery-powered hard disk. When power to the internal hard disk is turned off, the external hard disk could be turned off as well.

The structure of `HDQueueElement` is as follows.

```
typedef pascal void (*HDSpindownProc)(HDQueueElement *theElement);
struct HDQueueElement {
   Ptr              hdQLink;    /* pointer to next queue element */
   short            hdQType;    /* queue element type (must be HDQType) */
   short            hdFlags;    /* miscellaneous flags (reserved) */
   HDSpindownProc   hdProc;     /* pointer to routine to call */
   long             hdUser;     /* user-defined (variable storage, etc.) */
} HDQueueElement;
```

When power to the internal hard disk is about to be turned off, the software calls the routine pointed to by the `hdProc` field so that it can do any special processing. The software passes the routine a pointer to its queue element so that, for example, the routine can reference its variables.

Before calling `HardDiskQInstall`, the calling program must set the `hdQType` field to

```
#define HDPwrQType 'HD'      /* queue element type */
```

or the queue element won't be added to the queue and `HardDiskQInstall` will return an error.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskQInstall` is 10 ($0A) in the low word of register D0. The pointer to the `HDQueue` element is passed in register A0. The result code is returned in the low word of register D0.

HardDiskQRemove

You can use the `HardDiskQRemove` routine to discontinue notifying your software when power to the internal hard disk is about to be turned off.

```
OSErr HardDiskQRemove(HDQueueElement *theElement);
```

DESCRIPTION

The `HardDiskQRemove` routine removes a queue element installed by `HardDiskQInstall`. If the `hdQType` field of the queue element is not set to `HDPwrQType`, `HardDiskQRemove` simply returns an error.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `HardDiskQRemove` is 11 ($0B) in the low word of register D0. The pointer to the `HDQueue` element is passed in register A0. The result code is returned in the low word of register D0.

GetScaledBatteryInfo

You can use the `GetScaledBatteryInfo` routine to find out the condition of the battery or batteries.

```
void GetScaledBatteryInfo(short whichBattery, BatteryInfo *theInfo);
```

DESCRIPTION

The `GetScaledBatteryInfo` routine provides a generic means of returning information about the battery or batteries in the system. Instead of returning a voltage value, the routine returns the battery level as a fraction of the total possible voltage.

Note

New battery technologies such as NiCad (nickel cadmium) and nickel metal hydride (NiMH) have replaced the sealed lead acid batteries of the original Macintosh Portable. The algorithm for determining the battery voltage documented in the Power Manager chapter of *Inside Macintosh,* Volume VI, is no longer correct for all PowerBook models. ◆

The value of `whichBattery` determines whether `GetScaledBatteryInfo` returns information about a particular battery or about the total battery level. The value of `GetScaledBatteryInfo` should be in the range of 0 to `BatteryCount()`. If the value of `whichBattery` is 0, `GetScaledBatteryInfo` returns a summation of all the batteries, that is, the effective battery level of the whole system. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetScaledBatteryInfo` returns a result of 0 in all fields. Here is a summary of the effects of the `whichBattery` parameter:

Value of `whichBattery`	Information returned
0	Total battery level for all batteries
From 1 to `BatteryCount()`	Battery level for the selected battery
Less than 0 or greater than `BatteryCount`	0 in all fields of `theInfo`

The `GetScaledBatteryInfo` routine returns information about the battery in the following data structure:

```
typedef struct BatteryInfo {
   unsigned char     flags;            /* misc flags (see below) */
   unsigned char     warningLevel;     /* scaled warning level (0-255) */
   char              reserved;         /* reserved for internal use */
   unsigned char     batteryLevel;     /* scaled battery level (0-255) */
} BatteryInfo;
```

The `flags` character contains several bits that describe the battery and charger state. If a bit value is 1, that feature is available or is operating; if the bit value is 0, that feature is not operating. Unused bits are reserved by Apple for future expansion.

Field descriptions

Bit name	Bit number	Description
`batteryInstalled`	7	A battery is installed.
`batteryCharging`	6	The battery is charging.
`chargerConnected`	5	The charger is connected.

The value of `warningLevel` is the battery level at which the first low battery warning message will appear. The routine returns a value of 0 in some cases when it's not appropriate to return the warning level.

The value of `batteryLevel` is the current level of the battery. A value of 0 represents the voltage at which the Power Manager will force the computer into sleep mode; a value of 255 represents the highest possible voltage.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetScaledBatteryInfo` is 12 ($0C) in the low word of register D0. The `BatteryInfo` data are returned in the low word of register D0 as follows:

Bits 31–24	Flags
Bits 23–16	Warning level
Bits 15–8	Reserved
Bits 7–0	Battery level

AutoSleepControl

You can use the `AutoSleepControl` routine to turn the automatic sleep feature on and off.

```
void AutoSleepControl(Boolean enableSleep);
```

DESCRIPTION

The `AutoSleepControl` routine enables or disables the automatic sleep feature that causes the computer to go into sleep mode after a preset period of time. When `enableSleep` is set to `true`, the automatic sleep feature is enabled (this is the normal state). When `enableSleep` is set to `false`, the computer will not go into the sleep mode unless it is forced to either by some user action—for example, by the user's selecting Sleep from the Special menu of the Finder—or in a low battery situation.

IMPORTANT

Calling `AutoSleepControl` with `enableSleep` set to `false` multiple times increments the auto sleep disable level so that it requires the same number of calls to `AutoSleepControl` with `enableSleep` set to `true` to reenable the auto sleep feature. If more than one piece of software makes this call, auto sleep may not be reenabled when you think it should be. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `AutoSleepControl` is 13 ($0D) in the low word of register D0. The Boolean value is passed in the high word of register D0.

GetIntModemInfo

You can use the `GetIntModemInfo` routine to find out information about the internal modem.

```
unsigned long GetIntModemInfo();
```

DESCRIPTION

The `GetIntModemInfo` routine returns a 32-bit field containing information that describes the features and state of the internal modem. It can be called whether or not a modem is installed and will return the correct information.

If a bit is set, that feature or state is supported or selected; if the bit is cleared, that feature is not supported or selected. Undefined bits are reserved by Apple for future expansion.

Field descriptions

Bit name	Bit number	Description
`hasInternalModem`	0	An internal modem is installed.
`intModemRingDetect`	1	The modem has detected a ring on the telephone line.
`intModemOffHook`	2	The internal modem has taken the telephone line off hook (that is, you can hear the dial tone or modem carrier).
`intModemRingWakeEnb`	3	The computer will come out of sleep mode if the modem detects a ring on the telephone line and the computer supports this feature (see the `canWakeupOnRing` bit in `PMFeatures`).

Bit name	Bit number	Description
`extModemSelected`	4	The external modem is selected (if this bit is set, then the modem port will be connected to port A of the SCC; if the modem port is not shared by the internal modem and the SCC, then this bit can be ignored).

Bits 15–31 contain the modem type, which will take on one of the following values:

	–1	Modem is installed but type not recognized.
	0	No modem is installed.
	1	Modem is a serial modem.
	2	Modem is a PowerBook Duo–style Express Modem.
	3	Modem is a PowerBook 160/180–style Express Modem.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetIntModemInfo` is 14 ($0E) in the low word of register D0. The bit field to set is passed in the high word of register D0.

SetIntModemState

You can use the `SetIntModemState` routine to set some parts of the state of the internal modem.

```
void SetIntModemState(short theState);
```

DESCRIPTION

The `SetIntModemState` routine configures some of the internal modem's state information. Currently the only items that can be changed are the internal/external modem selection and the wakeup-on-ring feature.

To change an item of state information, the calling program sets the corresponding bit in `theState`. In other words, to change the internal/external modem setting, set bit 4 of `theState` to 1. To select the internal modem, bit 15 should be set to 0; to select the external modem, bit 15 should be set to 1. Using this method, the bits may be set or cleared independently, but they may not be set to different states at the same time.

Note
In some PowerBook computers, there is a hardware switch to connect either port A of the SCC or the internal modem to the modem port. The two are physically separated, but software emulates the serial port interface for those applications that don't use the Communications Toolbox. You can check the `hasSharedModemPort` bit returned by `PMFeatures` to determine which way the computer is set up. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetIntModemState` is 15 ($0F) in the low word of register D0. The bit field is returned in register D0.

MaximumProcessorSpeed

You can use the `MaximumProcessorSpeed` routine to find out the maximum speed of the computer's microprocessor.

```
short MaximumProcessorSpeed();
```

DESCRIPTION

The `MaximumProcessorSpeed` routine returns the maximum clock speed of the computer's microprocessor, in MHz.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `MaximumProcessorSpeed` is 16 ($10) in the low word of register D0. The processor speed value is returned in the low word of register D0.

CurrentProcessorSpeed

You can use the `CurrentProcessorSpeed` routine to find out the current clock speed of the microprocessor.

```
short CurrentProcessorSpeed();
```

DESCRIPTION

The `CurrentProcessorSpeed` routine returns the current clock speed of the computer's microprocessor, in MHz. The value returned is different from the maximum processor speed if the computer has been configured to run with a reduced processor speed to conserve power.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `CurrentProcessorSpeed` is 17 ($11) in the low word of register D0. The processor speed value is returned in the low word of register D0.

FullProcessorSpeed

You can use the `FullProcessorSpeed` routine to find out whether the computer will run at full speed the next time it restarts.

```
Boolean FullProcessorSpeed();
```

DESCRIPTION

The `FullProcessorSpeed` routine returns a Boolean value of `true` if, on the next restart, the computer will start up at its maximum processor speed; it returns `false` if the computer will start up at its reduced processor speed.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `FullProcessorSpeed` is 18 ($12) in the low word of register D0. The Boolean result is returned in the low byte of register D0.

SetProcessorSpeed

You can use the `SetProcessorSpeed` routine to set the clock speed the microprocessor will use the next time the computer is restarted.

```
Boolean SetProcessorSpeed(Boolean fullSpeed);
```

DESCRIPTION

The `SetProcessorSpeed` routine sets the processor speed that the computer will use the next time it is restarted. If the value of `fullSpeed` is set to `true`, the processor will start up at its full speed (the speed returned by `MaximumProcessorSpeed`, described on page 100). If the value of `fullSpeed` is set to `false`, the processor will start up at its reduced speed.

For PowerBook models that support changing the processor speed dynamically, the processor speed will also be changed. If the speed is actually changed, `SetProcessorSpeed` will return `true`; if the speed isn't changed, it will return `false`.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetProcessorSpeed` is 19 ($13) in the low word of register D0. The Boolean value to set is passed in the high word of register D0. The Boolean result is returned in register D0.

GetSCSIDiskModeAddress

You can use the `GetSCSIDiskModeAddress` routine to find out the SCSI ID the computer uses in SCSI disk mode.

```
short GetSCSIDiskModeAddress();
```

DESCRIPTION

The `GetSCSIDiskModeAddress` routine returns the SCSI ID that the computer uses when it is started up in SCSI disk mode. The returned value is in the range 1 to 6.

Note
When the computer is in SCSI disk mode, the computer appears as a hard disk to another computer. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetSCSIDiskModeAddress` is 20 ($14) in the low word of register D0. The SCSI ID is returned in the low word of register D0.

SetSCSIDiskModeAddress

You can use the `SetSCSIDiskModeAddress` routine to set the SCSI ID for the computer to use in SCSI disk mode.

```
void SetSCSIDiskModeAddress(short scsiAddress);
```

DESCRIPTION

The `SetSCSIDiskModeAddress` routine sets the SCSI ID that the computer will use if it is started up in SCSI disk mode.

The value of `scsiAddress` must be in the range of 1 to 6. If any other value is given, the software sets the SCSI ID for SCSI disk mode to 2.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetSCSIDiskModeAddress` is 21 ($15) in the low word of register D0. The SCSI ID to set is passed in the high word of register D0.

GetWakeupTimer

You can use the `GetWakeupTimer` routine to find out when the computer will wake up from sleep mode.

```
void GetWakeupTimer(WakeupTime *theTime);
```

DESCRIPTION

The `GetWakeupTimer` routine returns the time when the computer will wake up from sleep mode.

If the PowerBook model doesn't support the wakeup timer, `GetWakeupTimer` returns a value of 0. The time and the enable flag are returned in the following structure:

```
typedef struct WakeupTime {
   unsigned long  wakeTime;      /* wakeup time (same format as the time) */
   char           wakeEnabled;   /* 1=enable timer, 0=disable timer */
} WakeupTime;
```

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetWakeupTimer` is 22 ($16) in the low word of register D0. The pointer to `WakeupTime` is passed in register A0.

SetWakeupTimer

You can use the `SetWakeupTimer` routine to set the time when the computer will wake up from sleep mode.

```
void SetWakeupTimer(WakeupTime *theTime);
```

DESCRIPTION

The `SetWakeupTimer` routine sets the time when the computer will wake up from sleep mode and enables or disables the timer. On a PowerBook model that doesn't support the wakeup timer, `SetWakeupTimer` does nothing.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `SetWakeupTimer` is 23 ($17) in the low word of register D0. The pointer to `WakeupTime` is passed in register A0.

IsProcessorCyclingEnabled

You can use the `IsProcessorCyclingEnabled` routine to find out whether processor cycling is enabled.

```
Boolean IsProcessorCyclingEnabled();
```

DESCRIPTION

The `IsProcessorCyclingEnabled` routine returns a Boolean value of `true` if processor cycling is currently enabled, or `false` if it is disabled.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `IsProcessorCyclingEnabled` is 24 ($18) in the low word of register D0. The Boolean result is returned in register D0.

EnableProcessorCycling

You can use the `EnableProcessorCycling` routine to turn the processor cycling feature on and off.

```
void EnableProcessorCycling(Boolean enable);
```

DESCRIPTION

The `EnableProcessorCycling` routine enables processor cycling if a value of `true` is passed in, and disables it if `false` is passed.

▲ **WARNING**
You should follow the advice of the `mustProcessorCycle` bit in the feature flags when turning processor cycling off. Turning processor cycling off when it's not recommended can result in hardware failures due to overheating. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `EnableProcessorCycling` is 25 ($19) in the low word of register D0. The Boolean value to set is passed in the high word of register D0.

BatteryCount

You can use the `BatteryCount` routine to find out how many batteries the computer supports.

```
short BatteryCount();
```

DESCRIPTION

The `BatteryCount` routine returns the number of batteries supported internally by the computer. The return value of `BatteryCount` may not be the same as the number of batteries currently installed.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `BatteryCount` is 26 ($1A) in the low word of register D0. The number of batteries supported is returned in the low word of register D0.

GetBatteryVoltage

You can use the `GetBatteryVoltage` routine to find out the battery voltage.

```
Fixed GetBatteryVoltage(short whichBattery);
```

DESCRIPTION

The `GetBatteryVoltage` routine returns the battery voltage as a fixed-point number.

The value of `whichBattery` should be in the range 0 to `BatteryCount()`–1. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetBatteryVoltage` will return a result of 0.0 volts.

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetBatteryVoltage` is 27 ($1B) in the low word of register D0. The battery number is passed in the high word of register D0. The 32-bit value of the battery voltage is returned in register D0.

GetBatteryTimes

You can use the `GetBatteryTimes` routine to find out about how much battery time remains.

```
void GetBatteryTimes (short whichBattery, BatteryTimeRec *theTimes);
```

DESCRIPTION

The `GetBatteryTimes` routine returns information about the time remaining on the computer's battery or batteries. The information returned has the following data structure:

```
typedef struct BatteryTimeRec {
   unsigned long expectedBatteryTime;  /* estimated time remaining */
   unsigned long minimumBatteryTime;   /* minimum time remaining */
   unsigned long maximumBatteryTime;   /* maximum time remaining */
   unsigned long timeUntilCharged;     /* time until full charge */
} BatteryTimeRec;
```

The time values are in seconds. The value of `expectedBatteryTime` is the estimated time remaining based on current usage patterns. The values of `minimumBatteryTime` and `maximumBatteryTime` are worst-case and best-case estimates, respectively. The value of `timeUntilCharged` is the time that remains until the battery or batteries are fully charged.

The value of `whichBattery` determines whether `GetBatteryTimes` returns the time information about a particular battery or the total time for all batteries. The value of `GetScaledBatteryInfo` should be in the range of 0 to `BatteryCount()`. If the value of `whichBattery` is 0, `GetBatteryTimes` returns a total time for all the batteries, that is, the effective battery time for the whole system. If the value of `whichBattery` is out of range, or the selected battery is not installed, `GetBatteryTimes` returns a result of 0 in all fields. Here is a summary of the effects of the `whichBattery` parameter:

Value of `whichBattery`	Information returned
0	Total battery time for all batteries
From 1 to `BatteryCount( )`	Battery time for the selected battery
Less than 0 or greater than `BatteryCount`	0 in all fields of `theTimes`

ASSEMBLY-LANGUAGE INFORMATION

The trap is `_PowerMgrDispatch` ($A09E). The selector value for `GetBatteryTimes` is 28 ($1C) in the low word of register D0. The pointer to `BatteryTimeRec` is passed in register A0.

Header File for Power Manager Dispatch

Here is a sample header file for access to the Power Manager.

```
/************************************************************************************

   file:  PowerMgrDispatch.h

   contains: header file for access to the Power Manager

   Copyright © 1992-1993 by Apple Computer, Inc. All rights reserved.

************************************************************************************/

#ifndef __PowerMgrDispatch__

#define __PowerMgrDispatch__

#ifndef __TYPES__

#include <Types.h>

#endif


#ifndef gestaltPMgrDispatchExists

#define gestaltPMgrDispatchExists    4  /* gestaltPowerMgrAttr bit:
                                          1=PowerMgrDispatch exists */

#endif


/* bits in bitfield returned by PMFeatures */

#define hasWakeupTimer          0   /* 1=wakeup timer is supported  */

#define hasSharedModemPort      1   /* 1=modem port shared by SCC and internal modem */

#define hasProcessorCycling     2   /* 1=processor cycling is supported */

#define mustProcessorCycle      3   /* 1=processor cycling should not be turned off */

#define hasReducedSpeed         4   /* 1=processor can be started up at reduced speed */

#define dynamicSpeedChange      5   /* 1=processor speed can be switched dynamically */

#define hasSCSIDiskMode         6   /* 1=SCSI disk mode is supported */

#define canGetBatteryTime       7   /* 1=battery time can be calculated */

#define canWakeupOnRing    8   /* 1=can wake up when the modem detects a ring  */
```

```
/* bits in bitfield returned by GetIntModemInfo and set by SetIntModemState */
#define hasInternalModem   0  /* 1=internal modem installed */
#define intModemRingDetect 1  /* 1=internal modem has detected a ring */
#define intModemOffHook    2  /* 1=internal modem is off hook */
#define intModemRingWakeEnb3  /* 1=wake up on ring is enabled */
#define extModemSelected   4  /* 1=external modem selected */
#define modemSetBit        15 /* 1=set bit, 0=clear bit (SetIntModemState) */


/* information returned by GetScaledBatteryInfo */
struct BatteryInfo {
   unsigned charflags;           /* misc flags (see below) */
   unsigned charwarningLevel;    /* scaled warning level (0-255) */
   char   reserved;              /* reserved for internal use */
   unsigned charbatteryLevel;    /* scaled battery level (0-255) */
};


typedef struct BatteryInfo BatteryInfo;
/* bits in BatteryInfo.flags */
#define batteryInstalled   7     /* 1=battery is currently connected */
#define batteryCharging    6     /* 1=battery is being charged */
#define chargerConnected   5     /* 1=charger is connected to the PowerBook */
                                 /* (this doesn't mean the charger is plugged in) */


/* hard disk spindown notification queue element */
typedef struct HDQueueElement HDQueueElement;
typedef pascal void (*HDSpindownProc)(HDQueueElement *theElement);

struct HDQueueElement {

   Ptr           hdQLink;     /* pointer to next queue element */

   short         hdQType;     /* queue element type (must be HDQType) */

   short         hdFlags;     /* miscellaneous flags */

   HDSpindownProc hdProc;     /* pointer to routine to call */

   long          hdUser;      /* user-defined (variable storage, etc.) */

};


#define HDPwrQType'HD'        /* queue element type */

/* wakeup time record */

typedef struct WakeupTime {

   unsigned long    wakeTime;    /* wakeup time (same format as current time) */

   char             wakeEnabled; /* 1=enable wakeup timer, 0=disable wakeup timer */

} WakeupTime;


/* battery time information (in seconds) */

typedef struct BatteryTimeRec {

   unsigned long    expectedBatteryTime;   /* estimated battery time remaining */

   unsigned long    minimumBatteryTime;    /* minimum battery time remaining */

   unsigned long    maximumBatteryTime;    /* maximum battery time remaining */

   unsigned long    timeUntilCharged;      /* time until battery is fully charged */

} BatteryTimeRec;


#ifdef __cplusplus

extern "C" {

#endif
```

```
#pragma parameter __D0 PMSelectorCount(__D0)
short PMSelectorCount()
   = {0x7000, 0xA09E};

#pragma parameter __D0 PMFeatures
unsigned long PMFeatures()
   = {0x7001, 0xA09E};

#pragma parameter __D0 GetSleepTimeout
unsigned char GetSleepTimeout()
   = {0x7002, 0xA09E};

#pragma parameter __D0 SetSleepTimeout(__D0)
void SetSleepTimeout(unsigned char timeout)
   = {0x4840, 0x303C, 0x0003, 0xA09E};

#pragma parameter __D0 GetHardDiskTimeout
unsigned char GetHardDiskTimeout()
   = {0x7004, 0xA09E};

#pragma parameter __D0 SetHardDiskTimeout(__D0)
void SetHardDiskTimeout(unsigned char timeout)
   = {0x4840, 0x303C, 0x0005, 0xA09E};

#pragma parameter __D0 HardDiskPowered
Boolean HardDiskPowered()
   = {0x7006, 0xA09E};
```

```
#pragma parameter __D0 SpinDownHardDisk
void SpinDownHardDisk()
   = {0x7007, 0xA09E};

#pragma parameter __D0 IsSpindownDisabled
Boolean IsSpindownDisabled()
   = {0x7008, 0xA09E};

#pragma parameter __D0 SetSpindownDisable(__D0)
void SetSpindownDisable(Boolean setDisable)
   = {0x4840, 0x303C, 0x0009, 0xA09E};

#pragma parameter __D0 HardDiskQInstall(__A0)
OSErr HardDiskQInstall(HDQueueElement *theElement)
   = {0x700A, 0xA09E};

#pragma parameter __D0 HardDiskQRemove(__A0)
OSErr HardDiskQRemove(HDQueueElement *theElement)
   = {0x700B, 0xA09E};

#pragma parameter __D0 GetScaledBatteryInfo(__D0,__A0)
void GetScaledBatteryInfo(short whichBattery, BatteryInfo *theInfo)
   = {0x4840, 0x303C, 0x000C, 0xA09E, 0x2080};

#pragma parameter __D0 AutoSleepControl(__D0)
void AutoSleepControl(Boolean enableSleep)
   = {0x4840, 0x303C, 0x000D, 0xA09E};
```

```
#pragma parameter __D0 GetIntModemInfo(__D0)
unsigned long GetIntModemInfo()
   = {0x700E, 0xA09E};

#pragma parameter __D0 SetIntModemState(__D0)
void SetIntModemState(short theState)
   = {0x4840, 0x303C, 0x000F, 0xA09E};

#pragma parameter __D0 MaximumProcessorSpeed
short MaximumProcessorSpeed()
   = {0x7010, 0xA09E};

#pragma parameter __D0 CurrentProcessorSpeed
short CurrentProcessorSpeed()
   = {0x7011, 0xA09E};

#pragma parameter __D0 FullProcessorSpeed
Boolean FullProcessorSpeed()
   = {0x7012, 0xA09E};

#pragma parameter __D0 SetProcessorSpeed(__D0)
Boolean SetProcessorSpeed(Boolean fullSpeed)
   = {0x4840, 0x303C, 0x0013, 0xA09E};

#pragma parameter __D0 GetSCSIDiskModeAddress
short GetSCSIDiskModeAddress()
   = {0x7014, 0xA09E};
```

```
#pragma parameter __D0 SetSCSIDiskModeAddress(__D0)
void SetSCSIDiskModeAddress(short scsiAddress)
   = {0x4840, 0x303C, 0x0015, 0xA09E};

#pragma parameter __D0 GetWakeupTimer(__A0)
void GetWakeupTimer(WakeupTime *theTime)
   = {0x7016, 0xA09E};

#pragma parameter __D0 SetWakeupTimer(__A0)
void SetWakeupTimer(WakeupTime *theTime)
   = {0x7017, 0xA09E};

#pragma parameter __D0 IsProcessorCyclingEnabled
Boolean IsProcessorCyclingEnabled()
   = {0x7018, 0xA09E};

#pragma parameter __D0 EnableProcessorCycling(__D0)
void EnableProcessorCycling(Boolean enable)
   = {0x4840, 0x303C, 0x0019, 0xA09E};

#pragma parameter __D0 BatteryCount
short BatteryCount()
   = {0x701A, 0xA09E};

#pragma parameter __D0 GetBatteryVoltage(__D0)
Fixed GetBatteryVoltage(short whichBattery)
   = {0x4840, 0x303C, 0x001B, 0xA09E};
```

```
#pragma parameter __D0 GetBatteryTimes(__D0,__A0)
void GetBatteryTimes(BatteryTimeRec *theTimes)
   = {0x4840, 0x303C, 0x001C, 0xA09E};


#ifdef __cplusplus
}
#endif
#endif
```

C H A P T E R 8

Software for ATA Devices

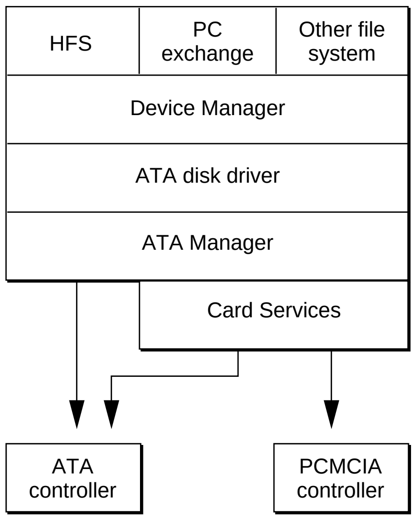
This chapter describes the system software that controls ATA devices in the Macintosh PowerBook 5300 computer. To use the information in this chapter, you should already be familiar with writing programs for the Macintosh computer that call device drivers to manipulate devices directly. You should also 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 5300 computer, the ATA software supports not only the internal ATA (IDE) 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 PC Exchange system extension.

The ATA software in the Macintosh PowerBook 5300 computer conforms to the Macintosh driver model. File systems communicate with the driver by way of the Device Manager, as shown in Figure 8-1. 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. See Chapter 9, "PC Card Services."

Figure 8-1 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 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 5300 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 HFS hard disk and floppy disk formats.
- Supports 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 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 PC Exchange and the new Drive Setup application. The functions are described in "ATA Disk Driver Reference" beginning on page 120.

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 Special menu of the Finder 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 in the Macintosh PowerBook 5300 computer does not support ejection of removable media in PC card drives. It supports drives such 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 pile 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 5300 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 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 with one or more drives is inactive. The spindown time, which can be set from the PowerBook control panel, is the same for all drives.

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 hard 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 9, "PC Card Services." ◆

The functions and data structures of the ATA Manager are described in "ATA Manager Reference" beginning on page 135.

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 122 and "Status Functions" beginning on page 130.

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*. The ATA disk driver also implements several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh Computers*. The control functions are listed in Table 8-1 and described in "Control Functions" beginning on page 122.

Table 8-1 Control functions

Value of csCode	Definition
5	Verify media
6	Format media
7	Eject drive
17	Enable or disable physical I/O access
21	Get drive icon
22	Get media icon

continued

Table 8-1 Control functions (continued)

Value of `csCode`	Definition
23	Drive information
44	Set startup partition
45	Set partition mounting
46	Set partition write protect
48	Clear partition mounting
49	Clear partition write protection
50	Register partition
51	Add a new drive to the drive queue
60	Mount volume
65	Driver-specific need-time code (system task time)
70	Power-mode status management control

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*. The ATA disk driver also implements several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh Computers*. The status functions are listed in Table 8-2 and described in "Status Functions" beginning on page 130.

Table 8-2 Status functions

Value of csCode	Definition
8	Return drive status information
43	Return driver Gestalt information
44	Return partition boot status
45	Return partition mount status
46	Return partition write protect status
51	Return partition information
70	Power mode status information

RESULT CODES

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

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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred.
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	The specified logical drive number does not exist
`offLinErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	The specified logical drive number does not exist

register partition

The `register partition` function supports 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

→	`csCode`	A value of 50.
→	`ioVRefNum`	The logical drive number.
→	`csParam[0–1]`	Pointer to new driver information.
←	`ioResult`	See result codes.

RESULT CODES

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

get a drive

The `get a drive` function supports 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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`controlErr`	Unimplemented control call; could not complete requested operation
`nsDrvErr`	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, or 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

→	`csCode`	A value of 70.
→	`ioVRefNum`	The logical drive number.
→	`csParam[0]`	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)
←	`ioResult`	See result codes.

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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 driver gestalt parameter block:

```
struct DriverGestaltParam
{
   ataPBHdr                                  // See definition on page 136
   SInt16            ioVRefNum;              // refNum of device
   SInt16            csCode;                 // Driver Gestalt code
   OSType            driverGestaltSelector;  // Gestalt selector
   driverGestaltInfo driverGestaltResponse;  // Returned result
};
typedef struct DriverGestaltParam DriverGestaltParam;
```

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

Parameter block

→	`csCode`	A value of 43.
→	`ioVRefNum`	The logical drive number.
→	`driverGestaltSelector`	Gestalt function selector. This is a 32-bit ASCII field containing one of the following selectors:

`'sync'`	Indicates synchronous or asynchronous driver.
`'devt'`	Specify type of device the driver is controlling.
`'intf'`	Specify the device interface.
`'boot'`	Specify PRAM value to designate this driver or device.
`'vers'`	Specify the version number of the driver.
`'lpwr'`	Indicates support for low-power mode.
`'dAPI'`	Indicates support for calls to PC Exchange.
`'purg'`	Indicates driver can be closed or purged.
`'wide'`	Indicates large volume support.
`'ejec'`	Eject call requirements.

←	`driverGestaltResponse`	Return value based on the driver gestalt selector. The possible return values are:

`'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): bit 0: if set, don't issue eject call on Restart. bit 1: if set, don't issue eject call on Shutdown.

←	`ioResult`	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, 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), 0 if it is not. 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 46.
→	`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 information

The `get partition information` function supports 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

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

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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 129.

Parameter block

→	`csCode`	A value of 70.
→	`ioVRefNum`	The logical drive number.
→	`csParam[]`	None defined.
←	`csParam[]`	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)
←	`ioResult`	See result codes.

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	The specified logical drive number does not exist
`statusErr`	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 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 depends 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 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.

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 Ptr 1
   Ptr      ataPtr2;       // Client storage Ptr 2
   UInt16   ataState;      // Reserved, init to 0
   SInt16   intSemaphores; // internal semaphores
   Sint32   hdrReserved4;  // Reserved
};
typedef struct ataPBHdr ataPBHdr;
```

Field descriptions

`ataLink`	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.
`ataQType`	This field is the queue type byte. It should be initialized to 0 before calling the ATA Manager.
`ataPBVers`	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 `paramErr`. For individual differences between versions 1 and 2, refer to the individual functions.
`hdrReserved`	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.
`hdrReserved2`	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.
`ataCompletion`	This field contains the completion routine pointer to be called upon completion of the request. When this field is set to 0, it indicates a synchronous I/O request; a non-zero 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:

```
pascal void (*RoutinePtr) (ataIOPB *)
```

The completion routine is called with the associated manager parameter block in the stack.

`ataResult`	Completion status. This field is returned by the ATA Manager after the request has been completed. Refer to Table 8-13 on page 175 for a list of the possible error codes returned in this field.
`MgrFCode`	This field is the function selector for the ATA Manager. The functions are defined in Table 8-4 on page 141. An invalid code in this field results in an `ATAFuncNotSupported` error.
`ataIOSpeed`	This field specifies the I/O cycle timing requirement of the specified ATA drive. This field should contain word 51 of the identify drive 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 identify drive data returned by the `ATA_Identify` function, the client should request operations using the slowest mode (mode 0).
`ataFlags`	This 16-bit field contains control settings that indicate special handling of the requested function. The control bits are defined in Table 8-3 on page 138.
`hdrReserved3`	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.
`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 manager.
`intSemaphores`	This field is used internally by the ATA Manager. It should be set to 0 before calling the ATA Manager.
`hdrReserved4`	Field reserved for future use. To ensure future compatibility, all reserved fields should be set to 0.

Table 8-3 describes the functions of the control bits in the `ataFlags` field.

Table 8-3 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.
`RegUpdate`	3	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 `ATA_ExecI/O` function only. Refer to the description on page 149 for details. The following device registers are reported back: ■ Sector count register ■ Sector number register ■ Cylinder register(s) ■ SDH register

continued

Table 8-3 Control bits in the `ataFlags` field (continued)

Name	Bit	Definition
`ProtocolType`	4–5	These two bits specify the type of command. The following command types are defined: $0 = standard ATA $1 = PCMCIA / ATA $2 = ATAPI These bits are used to indicate special protocol handling. 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.
—	6–7	Reserved.
`SGType`	8, 9	This 2-bit field specifies the type of scatter gather list passed in. This field is only valid for read / write operations. The following types are defined: 00 = scatter gather disabled 01 = scatter gather type I enabled 10 = reserved 11 = reserved When set to 0, this field indicates that the `ioBuffer` field contains the host buffer address for this transfer, and the `ioReqCount` field contains the byte transfer count. When set to 1, this field indicates that the `ioBuffer` and the `ioReqCount` 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. The format of the scatter gather list is a series of the following structure definition:

```
struct IOBlock       // SG entry structure
{
   UInt8* ioBuffer;  // → Data buffer pointer
   UInt32 ioReqCount;// → Byte count
};
typedef struct IOBlock IOBlock;
```

continued

Table 8-3 Control bits in the `ataFlags` field (continued)

Name	Bit	Definition
`QLockOnError`	10	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. 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.
`Immediate`	11	When this bit is set to 1, it indicates that the request must be executed as soon as possible and 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.
`ATAioDirection`	12, 13	This bit field specifies the direction of data transfer. Bit values are binary and defined as follows: 00 = no data transfer 10 = data direction in (read) 01 = data direction out (write) 11 = reserved Note: These bits do not need to be set to reflect the direction of the command packet bytes.
—	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 8-4.

Table 8-4 ATA Manager functions

Function name	Code	Description
`ATA_Abort`	$10	Terminate the command.
`ATA_BusInquiry`	$03	Get bus information.
`ATA_DrvrDeregister`	$87	Deregister the driver reference number.
`ATA_DrvrRegister`	$85	Register the driver reference number.
`ATA_ExecIO`	$01	Execute ATA I/O.
`ATA_EjectDrive`	$89	Auto-eject the drive.
`ATA_FindRefNum`	$86	Look up the driver reference number.
`ATA_GetDevConfig`	$8A	Get the device configuration.
`ATA_GetLocationIcon`	$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.
`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. Clean up includes parameter block deallocation and O/S 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 136
   ATA_PB*  AbortPB             // Address of the parameter
                                // block to be aborted
   UInt16   Reserved            // Reserved
};
typedef struct ATA_Abort ATA_Abort;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` parameter block on page 136.
`AbortPB`	This field contains the address of the I/O parameter block to be aborted.
`Reserved`	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
`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 gets 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 `ataBusInquiry` 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 136
   UInt16   ataEngineCount;     // ← TBD; zero for now
   UInt16   ataReserved;        // Reserved
   UInt32   ataDataTypes;       // ← TBD; zero for now
   UInt16   ataIOpbSize;          // ← Size of ATA I/O PB
   UInt16   ataMaxIOpbSize;       // ← TBD; zero for now
   UInt32   ataFeatureFlags;      // ← TBD
   UInt8    ataVersionNum;        // ← HBA Version number
   UInt8    ataHBAInquiry;        // ← TBD; zero for now
   UInt16   ataReserved2;         // Reserved
   UInt32   ataHBAPrivPtr;        // ← Ptr 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;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` on page 136.
`ataEngineCount`	Currently set to 0.
`ataReserved`	Reserved. All reserved fields are set to 0.
`ataDataTypes`	Returns a bit map 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_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 non-interrupt time.

The first driver to register for the device gets the device. 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 Section 6 of this document for possible events and their definition.

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 signalled by a value of –1 in the deviceID field of the header and the bit 0 of 'drvrFlags' 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 insure 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      ataDrvrRegister   // Parameter block structure
                              // for ataPBVers = 1
{
   ataPBHdr                   // Header information
   SInt16   drvrRefNum;       // → Driver reference number
   UInt16   FlagReserved;     // Reserved -> should be zero
   UInt16   deviceNextID;     // Not used
   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   drvrRefNum;       // → Driver reference number
   UInt16   drvrFlags;        // → Driver flags; set to 0
   UInt16   deviceNextID;     // Not used
   SInt16   Reserved;         // Reserved; set to zero
   ProcPtr ataEHandlerPtr     // → Event handler routine ptr
   SInt32   drvrContext;      // → 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 ataDrvrRegister ataDrvrRegister;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`drvrRefNum`	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.
`FlagReserved`	Reserved.
`deviceNextID`	Not used by this function.

`Reserved[21]`	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

`ataEHandlerPtr` 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 `ataEventMask` field is set. The calling convention for the event handler is as follows:

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

where the `ATAEventRec` is defined as follows:

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

See “Notification of Device Events” beginning on page 168 for a list of the ATA event codes.

`drvrContext` A value to be passed in when the event handler is called. This value will be loaded in the ATAEventRec before calling the event handler.

`ataEventMask` 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 only be called if the mask for the event has been set(1). If the mask is not set(0) for an event, the ATA Manager will take no action. Table 8-5 lists the masks have been defined.

Table 8-5 Event masks

Bits	**Event Mask**
$00	Null event
$01	Online event: a device has come on line
$02	Offline event: a device has gone off line
$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 off line
$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_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, which indicates that there is no driver 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)
structataDrvrRegister         // Parameter block structure
                              // for ataPBVers = 1
{
   ataPBHdr                   // Header information
   SInt16   drvrRefNum;       // Not used
   UInt16   FlagReserved;     // Reserved
   UInt16   deviceNextID;     // Not used
   SInt16   Reserved[21];     // Reserved for future expansion
};
typedef struct ataDrvrRegister ataDrvrRegister;

// Version 2(ataPBVers = 2)
structataDrvrRegister         // Parameter block structure
                              // for ataPBVers = 2
{
   ataPBHdr                   // Header information
   SInt16   drvrRefNum;       // → Driver reference number
   UInt16   drvrFlags;        // → driver flags; set to 0
   UInt16   deviceNextID;     // Not used
   SInt16   Reserved;         // Reserved -> should be zero
   ProcPtr ataEHandlerPtr     // → Event handler routine ptr
   SInt32   drvrContext;      // → 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 ataDrvrRegister ataDrvrRegister;
```

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
Note: Notify-all device drivers must deregister
using the parameter version two. ▲

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`drvrRefNum`	This field is not used with the deregister function.
drvrFlags	No bit definition has been defined for the field. This field shall be set to 0 in order to insure compatibility in the future.
deviceNextID	Not used for this function.
Reserved	Reserved. Should be set to 0
ataEHandlerPtr	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.
drvrContext	Not used for this function.
ataEventMask	Not used for this function.

RESULT CODES

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

ATA_EjectDrive

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

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 `ataPBHdr` parameter block definition on page 136.
`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 136
   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
```

`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 prior to 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.

`packetCDBPtr`	This field contains the packet pointer for ATAPI. The ATAPI bit of the ProtocolType 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 in order to insure compatibility in the future. 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 timeout 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)
structataDrvrRegister         // Parameter block structure
                              // for ataPBVers = 1
{
   ataPBHdr                   // Header information
   SInt16   drvrRefNum;       // ← 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)
structataDrvrRegister        // Parameter block structure
                             // for ataPBVers = 2
{
   ataPBHdr                  // Header information
   SInt16   drvrRefNum;      // ← Driver reference number
   UInt16   drvrFlags;       // → Reserved; set to 0
   UInt16   deviceNextID;    // ← used to specify the
                             // next drive ID
   SInt16   Reserved;        // Reserved -> should be zero
   ProcPtr  ataEHandlerPtr   // ← An event handler routine ptr
   SInt32   drvrContext;     // ← a 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;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`drvrRefNum`	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` 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_Get Device Configuration

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 data structure for the function is as follows:

```
struct      ataGetDevConfiguration// 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
                              // CardServices
   UInt8    socketType;       // ← Specifies the socket type
   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 vs. 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 136.
`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 8-6 on page 156 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.
`socketType`	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
`deviceType`	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
`pcAccessMode`	This field specifies the current access mode of the device; it is valid only if bit 0 of the pcValid field is set, and only for `ATA_GetDeviceConfiguration`, not for `ATA_SetDeviceConfiguration`. Possible values are: 0 = I/O mode 1 = Memory mode
`pcVcc`	This field indicates the current voltage setting of Vcc in tenths of a volt. It is valid only if bit 1 of the pcValid field is set.
`pcVpp1`	This field indicates the current voltage setting of Vpp1 in tenths of a volt. It is valid only if bit 2 of the pcValid field is set.
`pcVpp2`	This field indicates the current voltage setting of Vpp2 in tenths of a volt. It is valid only if bit 3 of the pcValid field is set.
`pcStatus`	This field indicates the current Card register setting of the PCMCIA device. It is valid only if bit 4 of the pcValid field is set.
`pcPin`	This field indicates the current Card Pin Register setting of the PCMCIA device. It is valid only if bit 5 of the pcValid field is set.

`pcCopy`	This field indicates the current Card Socket/Copy register setting of the PCMCIA device. It is valid only if bit 6 of the pcValid field is set.
`pcConfigIndex`	This field indicates the current Card Option register setting of the PCMCIA device. It is valid only if bit 7 of the pcValid field is set.

Table 8-6 Bits in pcValid field

Bits	Field validity indicated
0	pcAccessMode field is valid, when set
1	pcVcc field is valid, when set
2	pcVpp1 field is valid, when set
3	pcVpp2 field is valid, when set
4	pcStatus field is valid, when set
5	pcPin field is valid, when set
6	pcCopy field is valid, when set
7	pcConfigIndex field is valid, when set
8–14	Reserved (set to 0)
15	Reserved

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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 zero.

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 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 zero
   SInt8    *ataLocationIconPtr;
                             // → Pointer to icon data buffer
   SInt8    *ataLocationStringPtr;
                             // → Pointer to location string
                             // data buffer
   SInt16   Reserved[18];     // Reserved
};
typedef struct DrvLocationIcon DrvLocationIcon;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`ataIconType`	This field defines the type of icon desired as follows: $01 - large B&W icon with mask $81 - same as 1, but ProDOS icon
`ataIconReserved`	Reserved to be long-word aligned. This field should be set to zero for future compatibility.
`ataLocationIconPtr`	A pointer to the location icon buffer. When the pointer is non-zero, the function copies the icon data to the buffer.
`ataLocationStringPtr`	A pointer to the location string buffer. When the pointer is non-zero, the function copies the string data to the buffer.

RESULT CODES

`noErr`	Successful completion, no error occurred
`ATAInternalErr`	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 136
   SInt8    ataStatusReg;     // ← Last ATA status image
```

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 prior to the manager initialization, 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 136
   NumVersion MgrVersion     // ← Manager version number
   UInt8      MGRPBVers;     // ← Manager PB version number
                             // supported
   UInt8      Reserved1;     // Reserved
   UInt16     ataBusCnt;     // ← Number of ATA buses in system
   UInt16     ataDevCnt;     // ← Number of ATA devices detected
   UInt8      ataMaxMode;    // ← Maximum I/O speed mode
   UInt8      Reserved2;     // Reserved
   UInt16     IOClkResolution; // ← I/O clock resolution in nsec
   UInt16     Reserved[17];  // Reserved
};
typedef struct ATA_MgrInquiry ATA_MgrInquiry;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`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.
`Reserved`	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 a zero 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 a zero 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 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_ModifyDrvrEventMask

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

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

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 136.
`modifiedEventMask`	New event mask setting. The definitions of the subfields are given in Table 8-5 on page 146.
`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
`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. This function 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 136
   UInt16   Reserved[24];        // Reserved
};
typedef struct ataNOP ataNOP;
```

Field descriptions

ataPBHdr	See the definition of the `ataPBHdr` on page 136.

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 136
   UInt16   Reserved[24];        // Reserved
};
typedef struct ataQRelease ataQRelease;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` on page 136.

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.

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 136
   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;
   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 136
   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 Stat (R) or
                                 // Device Cntl (W) register
   UInt8    Reserved3;           // Reserved (set to 0)
   UInt16   Reserved[16];        // Reserved
};
typedef struct ataRegAccess ataRegAccess;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` parameter block on page 136.
`RegSelect`	This field specifies which of the device registers to access. The selectors for the registers supported by the `ATA_RegAccess` function are listed in Table 8-7.
`RegValue`	This field represents the value to be written (`ATAioDirection` = 01 binary) or the value read from the selected register (`ATAioDirection` = 10 binary). For the Data register, this field is a 16-bit field; for other registers, an 8-bit field. In the case where the `RegSelect` field is set to $FFFF (ataPBVers = 2 or higher), this field is sued to store the upper byte of the Data Register image.
`Reserved[2]`	This field is unused except in the cases where the RegSelect is set to either 0 (Data register) or $FFFF (more than one register selected). In those two cases, this field contains the lower byte of the Data register image.
`regMask`	This field is only valid for ataPBVers field 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 8-8.
`ri`	This field is only valid for `ataPBVers` field 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 regMask field are valid. Refer to 'ATA Execute I/O' section above for the structure definition.
`altStatDevCntrReg`	This field is only valid for `ataPBVers` 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 `regMask` field is set to one.

Table 8-7 ATA register selectors

Selector name	Selector	Register description
`DataReg`	0	Data register (16-bit access only)
`ErrorReg`	1	Error register (R) or features register (W)
`SecCntReg`	2	Sector count register
`SecNumReg`	3	Sector number register
`CylLoReg`	4	Cylinder low register
`CylHiReg`	5	Cylinder high register
`SDHReg`	6	SDH register
`StatusReg` `CmdReg`	7	Status register (R) or command register (W)
`AltStatus` `DevCntr`	$0E	Alternate status (R) or device control (W)
	$FFFF	More than one register access (valid only for ataPBVers = 2 or higher)

Table 8-8 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	SHD 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 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 136
   SInt8    Status;       // ← Last ATA status register image
   SInt8    Reserved;     // Reserved
   UInt16   Reserved[23]; // Reserved
};
typedef struct ATA_ResetBus ATA_ResetBus;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` parameter block on page 136.
`Status`	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.
`Reserved[23]`	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_SetDevConfiguration

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 data structure of the `ataSetDevConfiguration` 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;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 136.
`ConfigSetting`	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 (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 8-6 on page 156 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.
`pcAccessMode`	This field is valid only if the bit 0 of the pcValid field is set. The value is written to the access mode control. Possible values are: 0 = I/O mode 1 = Memory mode
`pcVcc`	This field indicates the new voltage setting of Vcc in tenths of a volt. It is valid only if the bit 1 of the pcValid field is set.
`pcVpp1`	This field indicates the new voltage setting of Vpp1 in tenths of a volt. It is valid only if the bit 2 of the pcValid field is set.
`pcVpp2`	This field indicates the new voltage setting of Vpp2 in tenths of a volt. It is valid only if the bit 3 of the pcValid field is set.
`pcStatus`	This field indicates the new Card Register setting of the PCMCIA device. It is valid only if the bit 4 of the pcValid field is set.
`pcPin`	This field indicates the new Card Pin Register setting of the PCMCIA device. It is valid only if the bit 5 of the pcValid field is set.

`pcCopy`	This field indicates the new Card Socket/Copy Register setting of the PCMCIA device. It is valid only if the bit 6 of the pcValid field is set.
`pcConfigIndex`	This field indicates the new Card Option Register setting of the PCMCIA device. It is valid only if the bit 7 of the pcValid field is set.

RESULT CODES

`noErr`	Successful completion, no error occurred
`nsDrvErr`	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 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 144 for the calling convention of the event handler.

The following event codes have been defined:

Table 8-9 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	Online event; signifies that a device has come online. This event may happen as a result of several actions: ▪ A device has been inserted into the socket. ▪ A device has been re-powered from sleep/low power.

continued

Table 8-9 Event codes send by the ATA Manager (continued)

Event code	Event description
	The client should let the O/S know about the presence of the device, if not done so already, verify the device type, and upload any device characteristics.
$02	Offline event; signifies that the device has gone offline. This event may happen as a result of several actions: ■ A device has been manually removed from the socket. The client should let the operating system know that the device has gone offline by setting the offline bit, if appropriate.
$03	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 O/S of the event.
$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 171.

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 re-initialization 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 8-10 shows the contents of the D5 register, high order bits first, for the old API (calls offset 0 bytes into the driver).

Table 8-10 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 8-11 shows the contents of the D5 register, high order bits first, for the new API. (calls offset 8 bytes into the driver)

Table 8-11 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 zero.
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 O/S
- partition name = `Apple_Driver_ATA`

The media driver should be capable of handling both old and new APIs. The 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 zero. 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 zero 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 driver). Refer to “Notification of Device Events” on page 168 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 zero indicates that the driver controls the device; a non-zero status indicates that the driver doesn't control the device.

The calling of the Notify-all drivers continues until a zero 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 non-zero status if the driver registration fails.

ROM Driver Notification

If no driver indicates that it controls the device, the ATA Manager calls the ATA HD driver in the system ROM. The ROM driver is called only for an HD device. For the Macintosh 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 permission: close(), DrvrRemove(), and DisposePtr()

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 = Do Close() and
                                    // DrvrRemove() this driver
                                    // refnum, but don't
                                    // deallocate driver code
                                    // 7 = Do Close(),
                                    // DrvrRemove(), and
                                    // DisposePtr()
   SInt16   purgeReserved;
   UniversalProcPtr purgeDrvrPointer;// <--: starting address
                                    // of the driver
                                    // (valid only if disposePtr
                                    // permission is given)
};
```

Setting the I/O Speed

The ATA controllers used in Macintosh systems have their I/O cycle time adjustable to optimize the 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 Set Socket Configuration function. The speed setting via the ATA Set Socket Configuration 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 Set Socket Configuration 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 'ATA Set Socket Configuration' 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 8-13 lists two sets of error codes for ATA drivers: old error codes, used with the Macintosh PowerBook 150 and the Macintosh 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 136. 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 8-13 ATA driver error codes

Error code (new)	Error code (old)	Error name	Error description
0	0	`noErr`	No error detected on the requested operation.
$FFCE (–50)	$FFCE (–50)	`paramErr`	Error in parameter block.
$FFC8 (–56)	$FFC8 (–56)	`nsDrvErr`	No such drive; no device is attached to the specified port.
$DB43 (–9405)	$F901 (–1791)	`AT_NRdyErr`	Drive ready condition not detected.
$DB44 (–9404)	$F904 (–1788)	`AT_IDNFErr`	Sector ID not found error reported by device.

continued

Table 8-13 ATA driver error codes (continued)

Error code (new)	Error code (old)	Error name	Error description
$DB45 (–9403)	$F905 (–1787)	`AT_DMarkErr`	Data mark not found reported by device.
$DB46 (–9402)	$F906 (–1786)	`AT_BadBlkErr`	Bad block detected by device.
$DB47 (–9401)	$F907 (–1785)	`AT_CorDataErr`	Notification that data was corrected (good data).
$DB48 (–9400)	$F906 (–1784)	`AT_UncDataErr`	Unable to correct data (possibly bad data).
$DB49 (–9399)	$F909 (–1783)	`AT_SeekErr`	Seek error detected by device.
$DB4A (–9398)	$F90A (–1782)	`AT_WrFltErr`	Write fault detected by device.
$DB4B (–9397)	$F90B (–1781)	`AT_RecalErr`	Recalibrate failure detected by device.
$DB4C (–9396)	$F90C (–1780)	`AT_AbortErr`	Command was aborted by device.
$DB4D (–9395)	$F90E (–1778)	`AT_MCErr`	Media-changed error.
$DB4E (–9394)	$F90F (–1777)	`ATAPICheckErr`	ATAPI Check Condition detected.
$DB70 (–9360)	$F8F6 (–1802)	`ATAMgrNotInitialized`	ATA Manager has not been initialized. The request function can not be performed until the manager has been initialized.
$DB71 (–9359)	$F8F5 (–1803)	`ATAPBInvalid`	Invalid ATA port address detected (ATA Manager initialization problem).
$DB72 (–9358)	$F8F4 (–1804)	`ATAFuncNotSupported`	An unknown ATA Manager function code specified.
$DB73 (–9357)	$F8F3 (–1805)	`ATABusy`	Selected device is busy; it is not ready to go to the next phase yet.
$DB74 (–9356)	$F8F2 (–1806)	`ATATransTimeOut`	Time-out condition 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.

continued

Table 8-13 ATA driver error codes (continued)

Error code (new)	Error code (old)	Error name	Error description
$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.
$DB7E (–9346)	$F8E8 (–1816)	`ATAInternalErr`	Card Services returned an error.
$DB7F (–9345)	$F8E7 (–1817)	`ATABusErr`	Bus error detected on I/O.
$DB80 (–9344)	$F90D (–1818)	`AT_NoAddrErr`	Invalid taskfile base address.
$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 DDM found on the media.
$DB93 (–9325)	—	`ATANoDriverErr`	No driver found on the media.

C H A P T E R 9

PC Card Services

This chapter describes the Card Services part of the PC Card Manager.

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 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
   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

`SUCCESS`	No error
`BAD_SOCKET`	Invalid socket specified
`NO_MORE_ITEMS`	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 180.

DESCRIPTION

The `CSGetNextClient` function returns the next `clientHandle` 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

`SUCCESS`	No error
`BAD_SOCKET`	Invalid socket specified
`NO_MORE_ITEMS`	No clients registered
`BAD_HANDLE`	Invalid `clientHandle`

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
{
   UInt32clientHandle;// → clientHandle returned by RegisterClient
   UInt16attributes; // ↔ 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;

      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
};
```

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 `clientHandle` 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,
   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 `clientHandle` 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 185.

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.

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

`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

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)
   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];   //
};
```

For `attributes`, `intType`, `present`, and `cardValues` field values see "CSGetConfigurationInfo" beginning on page 185.

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);
```

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];
};

// '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 189). 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);
```

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.

RESULT CODES

`SUCCESS`	No error
`BAD_HANDLE`	Invalid `clientHandle` 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 192.

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 `clientHandle` field of `GetClientInfoPB` 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 192.

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
   UInt16  eventMask;   // → bitmap of events to pass to client
                        //     for this socket
};
```

For `eventMask` field values, see “CSGetClientEventMask” on page 192.

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

   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 function
`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);
```

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 196.

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 unable 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
```

```
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

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,

   csExtAccSpeedExp1ns  = 0x00,
   csExtAccSpeedExp10ns = 0x01,
   csExtAccSpeedExp100ns= 0x02,
   csExtAccSpeedExp1us  = 0x03,
   csExtAccSpeedExp10us = 0x04,
   csExtAccSpeedExp100us= 0x05,
   csExtAccSpeedExp1ms  = 0x06,
   csExtAccSpeedExp10ms = 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

`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

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
   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 201.

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 201.

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`.

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
   PCCardCSClientUPP 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 206.

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);
```

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

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 `vsCode` 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
};
```

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

#define  csFixedDiskCardType     5
#define  csUnknownFixedDiskType  0
#define  csATAInterface          1
#define  csRotatingDevice        (0<<7)
#define  csSiliconDevice         (1<<7)
#define  csVideoAdaptorCardType  6

#define  csNetworkAdaptorCardType7
#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 CS`RequestSocketMask` 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,
   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 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
};

/*
   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
};

// 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
    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

680x0 code Instructions that can run on a PowerPC microprocessor only by means of an emulator. See also **native code.**

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.

Apple SuperDrive Apple Computer's disk drive for high-density floppy disks.

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.

big-endian Data formatting in which each field is addressed by referring to its most significant byte. See also **little-endian.**

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.

DAC See **digital-to-analog converter.**

data burst Multiple longwords of data sent over a bus in a single, uninterrupted stream.

data cache In a PowerPC microprocessor, the internal registers that hold data being processed.

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.**

DR Emulator The Dynamic Recompilation Emulator, an improved 680x0-code emulator for the PowerPC microprocessor.

dynamic random-access memory (DRAM) Random-access memory in which each storage address must be periodically interrogated ("refreshed") to maintain its value.

Ethernet A high-speed local area network technology that includes both cable standards and a series of communications protocols.

GCR See **group code recording.**

Grand Central A custom IC that provides core I/O services in second-generation Power Macintosh computers.

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.

mini-DIN An international standard form of cable connector for peripheral devices.

native code Instructions that run directly on a PowerPC microprocessor. See also **680x0 code.**

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.

NuBus adapter card A card for the Power Macintosh 6100/60 that gives the computer NuBus capability. It plugs into the PDS connector and accepts short NuBus cards.

PBX The custom IC that provides the interface between the PowerPC 603 bus and the I/O bus in a Macintosh PowerBook 5300 computer.

PC card An expansion card that conforms to the PCMCIA standard.

PC Card Manager The part of the Mac OS that supports PC cards in PowerBook computers.

PC Exchange A utility program that runs on Macintosh computers and reads other floppy disk formats, including DOS and ProDOS.

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-clean Refers to PowerPC code free of instructions that are specific to the PowerPC 601 and Power instruction sets and are not found on the PowerPC 603 and PowerPC 604 microprocessors.

PowerPC Trade name for a family of RISC microprocessors. The PowerPC 601, 603, and 604 microprocessors are used in Power Macintosh computers.

reduced instruction set computing (RISC) A technology of microprocessor design in which all machine instructions are uniformly formatted and are processed through the same steps.

RISC See **reduced instruction set computing.**

SCC See **Serial Communications Controller.**

SCSI See **Small Computer System Interface.**

Serial Communications Controller (SCC) Circuitry on the Combo IC that provides an interface to the serial data ports.

SIMM See **Single Inline Memory Module.**

Single Inline Memory Module (SIMM) A plug-in card for memory expansion, containing several RAM ICs and their interconnections.

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 that a PC Card is inserted into.

Socket Services The layer of software that is responsible for communication between Card Services and the socket controller hardware.

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 14

A

AC adapter 4
access to internal components 24
access windows 201
active matrix display 25
ADB (Apple Desktop Bus) port 27
ADB connector 27
appearance 3
Ariel CLUT-DAC IC 16
`ATA_Abort` function 141
`ATA_BusInquiry` function 142
ATA disk driver 117, 120–135
  `clear partition mounting` function 127
  `clear partition write protect` function 127
  control functions 122–130
  control routine 120
  Device Manager routines 120–122
  `driver gestalt` function 131
  `driverGestaltParam` parameter block 131
  `drive status` function 130
  `eject` function 123
  `format` function 123
  `get a drive` function 128
  `get drive icon` function 124
  `get drive information` function 125
  `get media icon` function 124
  `get partition information` function 134
  `get partition mount status` function 133
  `get partition write protect status` function 133
  `get power mode` function 134
  `mount volume` function 129
  `register partition` function 128
  `set partition mounting` function 126
  `set partition write protect` function 126
  `set power mode` function 129
  `set startup partition` function 125
  status functions 130–135
  status routine 121
  `verify` function 122
`ATA_DrvrDeregister` function 147
`ATA_DrvrRegister` function 144
`ATA_ExecIO` function 149
`ATA_FindRefNum` function 152
ATA hard disk drives, compared with SCSI drives 117
`ATA_Identify` function 157
ATA IDE specification 116
ATA interface 18
ATA Manager 135–177
  making calls to 135
  purpose of 117, 119
ATA Manager functions
  `ATA_Abort` 141
  `ATA_BusInquiry` 142
  `ATA_DrvrDeregister` 147
  `ATA_DrvrRegister` 144
  `ATA_ExecIO` 149
  `ATA_FindRefNum` 152
  `ATA_Identify` 157
  `ATA_MgrInquiry` 159
  `ATA_NOP` 161
  `ATA_QRelease` 161
  `ATA_RegAccess` 162
  `ATA_ResetBus` 165
`ATA_MgrInquiry` function 159
`ATA_NOP` function 161
ATA parameter block header 136
`ataPBHdr` structure 136–140
`ATA_QRelease` function 161
`ATA_RegAccess` function 162
`ATA_ResetBus` function 165
ATA software
  ATA disk driver 117
  ATA Manager 119
  error codes 175
`AutoSleepControl` routine 97

B

Baboon custom IC 15
back view 4
batteries 4
`BatteryCount` routine 105
`BlockCopy` routine 71
`BlockMoveData` routine 70
`BlockMoveDataUncached` routine 70
`BlockMove` extensions 69–70

BlockMove routine 70
BlockMoveUncached routine 70
BlockZero routine 70
BlockZeroUncached routine 70

C

cache coherency 7, 72
Card Services software 180–217
  access window functions
    CSModifyWindow 203
    CSReleaseWindow 204
    CSRequestWindow 201
  client information functions
    CSGetClientInfo 182
    CSGetFirstClient 180
    CSGetNextClient 181
  client registration functions
    CSDeregisterClient 207
    CSGetCardServicesInfo 205
    CSRegisterClient 206
  clients 180
  configuration functions
    CSAccessConfigurationRegister 191
    CSGetConfigurationInfo 185
    CSModifyConfiguration 189
    CSReleaseConfiguration 190
    CSRequestConfiguration 187
  gestalt constant 215
  masking functions
    CSGetClientEventMask 192
    CSReleaseSocketMask 195
    CSRequestSocketMask 194
    CSSetClientEventMask 193
  messages 216
  miscellaneous functions
    CSResetCard 208
    CSValidateCIS 209
    CSVendorSpecific 210
  result codes 215
  status functions, CSGetStatus 200
  tuples functions
    CSGetFirstTuple 196
    CSGetNextTuple 197
    CSGetTupleData 198
  unsupported functions
    CSReleaseExclusive 215
    CSRequestExclusive 215
  vendor-specific calls
    EjectCard 211
    EnableSocketEvents 213
    GetAdapterInfo 213
    GetCardInfo 211
clear partition mounting function 127
clear partition write protect function 127
clients, registration of 205
Code Fragment Manager 7, 72
Combo custom IC 13, 14
compatibility 5
  with the PowerPC 601 72, 73
  sound sample rates 6
configurations 5
connectors
  ADB 27
  expansion bay 33
  external video 51, 52
  hard disk 20
  RAM expansion 40, 42
  RAM expansion card 48
  SCSI 25
  serial port 25
  video 51
control routine 120
CSAccessConfigurationRegister function 191
CSDeregisterClient function 207
CSGetCardServicesInfo function 205
CSGetClientEventMask function 192
CSGetClientInfo function 182
CSGetConfigurationInfo function 185
CSGetFirstClient function 180
CSGetFirstTuple function 196
CSGetNextClient function 181
CSGetNextTuple function 197
CSGetStatus function 200
CSGetTupleData function 198
CSModifyConfiguration function 189
CSModifyWindow function 203
CSRegisterClient function 206
CSReleaseConfiguration function 190
CSReleaseExclusive function 215
CSReleaseSocketMask function 195
CSReleaseWindow function 204
CSRequestConfiguration function 187
CSRequestExclusive function 215
CSRequestSocketMask function 194
CSRequestWindow function 201
CSResetCard function 208
CSSetClientEventMask function 193
CSValidateCIS function 209
CSVendorSpecific function 210
CurrentProcessorSpeed routine 100
custom ICs
  Ariel 16
  Baboon 15
  Combo 14
  ECSC 14
  Keystone 16
  PBX 12

custom ICs *(continued)*
  Pratt 10
  Singer 14, 28
  TREX 15
  Whitney 13

D

`dcbz` instruction 70
Device Manager 79
display controller IC 14
Display Manager 74
displays
  active matrix 24, 25
  backlighting 24
  dual mode 50
  DualScan 25
  external video monitors 49, 50
    adapter cable 51
  flat panel types 24
  FSTN 25
  mirror mode 6, 50
  NuBus card emulation 24
  number of colors 6, 24
  passive matrix 24
  supertwist 24, 25
  TFT 25
`driver gestalt` function 131
`driverGestaltParam` parameter block 131
Driver Services Library 71
Drive Setup, modifications to 68
`drive status` function 130
dual mode 50
DualScan display 25
Dynamic Recompilation Emulator 68

E

ECSC custom IC 14
`EjectCard` vendor-specific call 211
`eject` function 123
Emulator, Dynamic Recompilation 68
`EnableProcessorCycling` routine 104
`EnableSocketEvents` vendor-specific call 213
error codes 175
Ethernet driver 64
event mask 192
expansion bay 32–39
  device installation 38
expansion bay connector 33–38
  signal assignments 34
  signal definitions 36–37
expansion bay controller IC 15
extended I/O parameter block 79–80
extended volume parameter block 77–79
external video port 50

F

features summary 2
Finder modifications for large volume support 67, 76
flat panel displays 24
`format` function 123
front view 3
`FullProcessorSpeed` routine 101
function-key software 64

G

`Gestalt` function 81
`gestaltMachineType` value 62
`gestaltPowerMgrAttr` selector 87
`GetAdapterInfo` vendor-specific call 213
`get a drive` function 128
`GetBatteryTimes` routine 106
`GetBatteryVoltage` routine 105
`get boot partition` function 132
`GetCardInfo` vendor-specific call 211
`get drive icon` function 124
`get drive information` function 125
`GetHardDiskTimeout` routine 91
`GetIntModemInfo` routine 98
`get media icon` function 124
`get partition information` function 134
`get partition mount status` function 133
`get partition write protect status` function 133
`get power mode` function 134
`GetScaledBatteryInfo` routine 95
`GetSCSIDiskModeAddress` routine 102
`GetSleepTimeout` routine 90
`GetWakeupTimer` routine 103

H

hard disk 18
  dimensions 18
  IDE data bus 22
hard disk capacity 5
hard disk connector 20
  pin assignments on 20
  signals on 21

`HardDiskPowered` routine 92
`HardDiskQInstall` routine 94
`HardDiskQRemove` routine 95
HDI-30 connector 25
HFS volume format 76

I, J

IDE disk interface 18
IDE hard disk 18
  connector 20
    pin assignments on 20
  data bus 22
  dimensions 18
  signals 21
identifying the computers 62
IDE specification 116
infrared module 28
input/output subsystem 10
interpretive emulator 68
I/O ports
  SCSI 25
  serial 25
  video 50, 51
`IsProcessorCyclingEnabled` routine 104
`IsSpindownDisabled` routine 93

K

keyboards 23
  function keys 64
  ISO layout 23
  removing 24
  United States layout 23
Keystone video timing IC 16

L

large partition support. *See* large volume support
large volume support 66, 76
  allocation blocks 76
  extended API 66
  extended data structures 77
  extended parameter block 77, 79
  limitations 68
  maximum file size 77
  modified applications 67
  requirements 77

M, N, O

main processor 11
`MaximumProcessorSpeed` routine 100
MC68LC040 microprocessor support 62
memory controller IC 12
memory controller software 63
memory expansion 4, 11
mirror mode 6, 50
monitor sense codes 52
`mount volume` function 129

P

PBX custom IC 12
  as bus bridge 12
`PBXGetVolInfo` function 81
PC cards 58, 180
  Finder extension for 59
  software eject 58
PCMCIA cards. *See* PC cards
PCMCIA slot 58–60
  access windows 59
  data access modes 59
  features 58
  power 60
  signal definitions 60
  specifications 59–60
PCMCIA slots 15
peripheral devices 4
peripheral support IC 13
`PMFeatures` routine 89
`PMSelectorCount` routine 89
pointing device 22
POWER-clean code 72
POWER-clean native code 71
POWER emulation 72
  exception handling 72
POWER instructions
  emulation of 72
Power Manager IC 14
  trackpad registers in 65
Power Manager interface routines 87–106
  `AutoSleepControl` 97
  `BatteryCount` 105
  `CurrentProcessorSpeed` 100
  `EnableProcessorCycling` 104
  `FullProcessorSpeed` 101
  `GetBatteryTimes` 106
  `GetBatteryVoltage` 105
  `GetHardDiskTimeout` 91
  `GetIntModemInfo` 98
  `GetScaledBatteryInfo` 95

Power Manager interface routines *(continued)*
`GetSCSIDiskModeAddress` 102
`GetSleepTimeout` 90
`GetWakeupTimer` 103
`HardDiskPowered` 92
`HardDiskQInstall` 94
`HardDiskQRemove` 95
`IsProcessorCyclingEnabled` 104
`IsSpindownDisabled` 93
`MaximumProcessorSpeed` 100
`PMFeatures` 89
`PMSelectorCount` 89
`SetHardDiskTimeout` 92
`SetIntModemState` 99
`SetProcessorSpeed` 101
`SetSCSIDiskModeAddress` 102
`SetSleepTimeout` 91
`SetSpindownDisable` 94
`SetWakeupTimer` 103
`SpinDownHardDisk` 93
Power Manager software 63, 86
checking for routines 87
data structures 6
dispatching 107
interface routines 86, 87–106
unsafe assumptions 7, 86
PowerPC 601 microprocessor 72
compatibility limitations 73
compatibility with 72
PowerPC 603 microprocessor 11, 71, 72
PowerPC 604 microprocessor 72
processor clock speed 2, 5, 11
processor/memory subsystem 10, 11

Q

QuickDraw acceleration API 73

R

RAM
contiguous banks of 12
expansion 4, 11, 39–48
addressing 43
DRAM devices 45
RAM banks 45
signals 40, 42
size of 5
RAM expansion card 39–48
connector 48
dimensions 47
DRAM devices 45
electrical limits 46
mechanical design of 47–48
RAM banks 45
reference documents xii
`register partition` function 128
Resource Manager in native code 69
ROM
address range 12
implementation of 12
software features 62
ROM software features 62

S

SCC IC 14
SCSI controller IC 14
SCSI port 25
connector 25
secondary logic board 11
serial port 25
`SetHardDiskTimeout` routine 92
`SetIntModemState` routine 99
`set partition mounting` function 126
`set partition write protect` function 126
`set power mode` function 129
`SetProcessorSpeed` routine 101
`SetSCSIDiskModeAddress` routine 102
`SetSleepTimeout` routine 91
`SetSpindownDisable` routine 94
`set startup partition` function 125
`SetWakeupTimer` routine 103
Singer custom IC 13, 14, 28
socket mask 192
sound
input sources 29
built-in microphone 29
CD-ROM drive 29
PCMCIA slot 29
sound input jack 29
output devices 29
sound circuits 29
characteristics 29, 30
sound features 63
sound IC 14, 28
sound sample rates 6, 28
sound specifications 28
`SpinDownHardDisk` routine 93
status routine 121
Supertwist display 25
System 7.5 66

T, U

TFT display 25
trackball 22
trackpad 22
  software support for 65
TREX custom IC 15
tuple information 196
tuples 119

V

VCB allocation block size 76
`verify` function 122
video adapter cable 51
video card 16, 49–58
video connector 51
video controller IC 16
video modes
  dual 50
  mirror 6, 50
video monitors 49, 50
  adapter cable for 51
  sense codes 52
  VGA and SVGA 52
video output IC 16
video port 50

W

Whitney custom IC 13

X, Y, Z

`XIOParam` data structure 79
`XVolumeParam` parameter block 77

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 EDITOR
Wendy Krafft

ILLUSTRATORS
Deb Dennis, Sandee Karr,
Shawn Morningstar

PRODUCTION EDITOR
Rex Wolf

Special thanks to Mark Baumwell, John Howard, Tom Llewellyn, Richard Schnell, Mark Seibert, George Towner, and Markus Wallgren