Developer Note

Macintosh PowerBook Duo 2300c Computer

Developer Press
© Apple Computer, Inc. 1995

Contents

Figures and Tables

About This Note

This developer note describes the Macintosh PowerBook Duo 2300c computer, emphasizing the features that are new or different from those of earlier PowerBook Duo computers. This developer note is a supplement to the *Macintosh PowerBook Duo Developer Note*, described in the section "Supplementary Documents," later in this preface.

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.

Contents of This Note

This developer note is arranged in seven chapters and an appendix:

- Chapter 1, "Introduction," describes the Macintosh PowerBook Duo 2300c computer and compares it with other PowerBook Duo models.
- Chapter 2, "Architecture," describes the architecture of the computer, with emphasis on the PowerPC 603 microprocessor and the custom ICs.
- Chapter 3, "Input and Output Features," describes the input and output features and the internal hard disk drive.
- Chapter 4, "Software Features," describes the software features that are specific to the Macintosh PowerBook Duo 2300c computer.
- Chapter 5, "Power Manager Interface," describes the application programming interface for the Power Manager software.
- Chapter 6, "Large Volume Support," describes the modifications that enable the file system to support volumes larger than 4 GB.
- Chapter 7, "Software for the ATA Hard Disk," describes the software that supports the internal IDE hard disk drive.
- The appendix, "Color Lookup Table," describes the table that determines the colors that appear on the built-in color display.

The chapters and appendix are followed by a glossary and an index.

Supplementary Documents

To supplement the information in this developer note, developers should have copies of the *PowerPC 601 RISC Microprocessor User's Manual* and *PowerPC 603 Microprocessor Implementation Definition Book IV.* These books are available from Motorola.

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

For information about the original Macintosh PowerBook Duo computers, developers should have a copy of the *Macintosh PowerBook Duo Developer Note*, available on Apple's Developer CD Series as well as through APDA (order *Macintosh Developer Notes, Number 2,* APDA catalog number R0457LL/A). Information about the PowerBook Duo 280 and 280c computers and the Duo Dock II is published in *Macintosh Developer Note Number 9*, APDA catalog number R0567LL/A.

For information about native drivers and the system registry, developers should have a copy of *Designing PCI Cards and Drivers for Power Macintosh Copmuters.*

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

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. 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

Conventions and Abbreviations

This developer note uses the following typographical conventions and abbreviations.

Typographical Conventions

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

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

A slash in front of a signal name (/RESET) indicates an active-low signal.

Note
A note like this contains information that is of interest but is not essential for an understanding of the text. ◆

Sidebar

A sidebar is 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.

IMPORTANT
A note like this contains information that is essential to an understanding of the text or of the computer. ▲

▲ **WARNING**
A note like this directs your attention to something that could cause injury to staff, damage to equipment, or loss of data. ▲

Abbreviations

When unusual abbreviations appear in this developer note, the corresponding terms are spelled out. Standard units of measure and other widely used abbreviations are not spelled out.

Standard units of measure used in this note include

GB	gigabytes
K	1024
KB	kilobytes
MB	megabytes
MHz	megahertz
nsec	nanoseconds
V	volts

Other abbreviations used in this note include

$*n*	hexadecimal value *n*
ADB	Apple Desktop Bus
ANSI	American National Standards Institute
API	application programming interface
CCFL	cold cathode fluorescent lamp
CLUT	color lookup table
CPU	central processing unit (the main microprocessor)
CRT	cathode ray tube (video display device)
CSC	color support chip (a custom IC)
DLPI	data link provider interface
DRAM	dynamic RAM

FSTN	film supertwist nematic (a type of LCD)
HBA	host bus adapter
HFS	hierarchical file system
IC	integrated circuit
IDE	integrated device electronics
I/O	input/output
LCD	liquid crystal display
LED	light-emitting diode
MMU	memory management unit
NiCad	nickel cadmium
NiMH	nickel metal hydride
PB API	parameter-block application program interface
PDS	processor-direct slot
POWER	performance optimized with enhanced RISC
PRAM	parameter RAM (nonvolatile RAM)
RAM	random-access memory
RAMDAC	random-access memory, digital-to-analog converter
RISC	reduced instruction set computing
ROM	read-only memory
SCC	Serial Communications Controller
SCSI	Small Computer System Interface
SRAM	static RAM
TFT	thin-film transistor (a type of LCD)
TPI	transport provider interface
VRAM	video RAM

C H A P T E R 1

Introduction

Introduction

The Macintosh PowerBook Duo 2300c computer is the first of a new generation of PowerBook Duo computers using the PowerPC™ 603 microprocessor. In addition to all the basic features of earlier PowerBook Duo models, the PowerBook Duo 2300c computer also has certain new features described in this developer note.

Features

The following summary of features constitutes a general description of the Macintosh PowerBook Duo 2300c computer. This computer has several new features that distinguish it from the earlier PowerBook Duo computers described in *Macintosh Developer Notes,Number 2*, and *Macintosh Developer Note Number 9*. The new features are described later in this developer note.

- **Processor.** The Macintosh PowerBook Duo 2300c computer has a PowerPC 603e microprocessor running at a clock frequency of 100 MHz.
- **RAM:** The built-in memory consists of 8 MB of low-power, self-refreshing dynamic RAM (DRAM).
- **RAM expansion:** The computer accepts a RAM expansion card with up to 48 MB, for a total of 56 MB of RAM.
- **Display:** The computer has a flat panel display with either 640-by-400 pixels and thousands of colors or 640-by-480 pixels and 256 colors. The display is an active-matrix LCD; it is backlit by a cold cathode fluorescent lamp (CCFL).
- **Hard disk:** The computer has one internal 2.5-inch IDE hard disk drive with disk capacity of either 750 MB or 1 GB. See "Configurations" on page 3.
- **SCSI disk mode:** With an optional HDI-30 SCSI Disk Adapter cable, the computer allows the user to read and store data on the computer's internal hard disk from another Macintosh computer.
- **Modem:** The computer accepts an Express Modem fax/modem card.
- **Networking:** The computer has a built-in LocalTalk network interface.
- **Sound:** The computer has a built-in microphone and speaker. The sound circuits provide 16-bit monaural sound input and output.
- **Keyboard:** The computer has an integral full-function keyboard with trackpad.
- **I/O ports:** The computer has one 152-pin connector for expansion devices, one mini-DIN 8-pin serial port, and one modem port.
- **Battery:** The computer uses a 4.5 ampere-hour removable and rechargeable nickel metal hydride (NiMH) battery.
- **Weight:** The computer weighs 2.2 kilograms (4.8 pounds) with the battery installed.
- **Size:** When the computer is closed, it measures 203.2 by 274.32 millimeters (8 by 10.8 inches) and is 36.8 millimeters (1.449 inches) deep.

Configurations

The Macintosh PowerBook Duo 2300c computer is available in two configurations, as shown in Table 1-1.

Table 1-1 Configurations

Amount of RAM	**Size of hard disk**	**Modem included**
8 MB	750 MB	No
8 MB	1 GB	Yes

Appearance

Figure 1-1 shows the Macintosh PowerBook Duo 2300c computer with its clamshell case in the open position. Figure 1-2 shows the back of the computer.

Figure 1-1 Front view of the computer

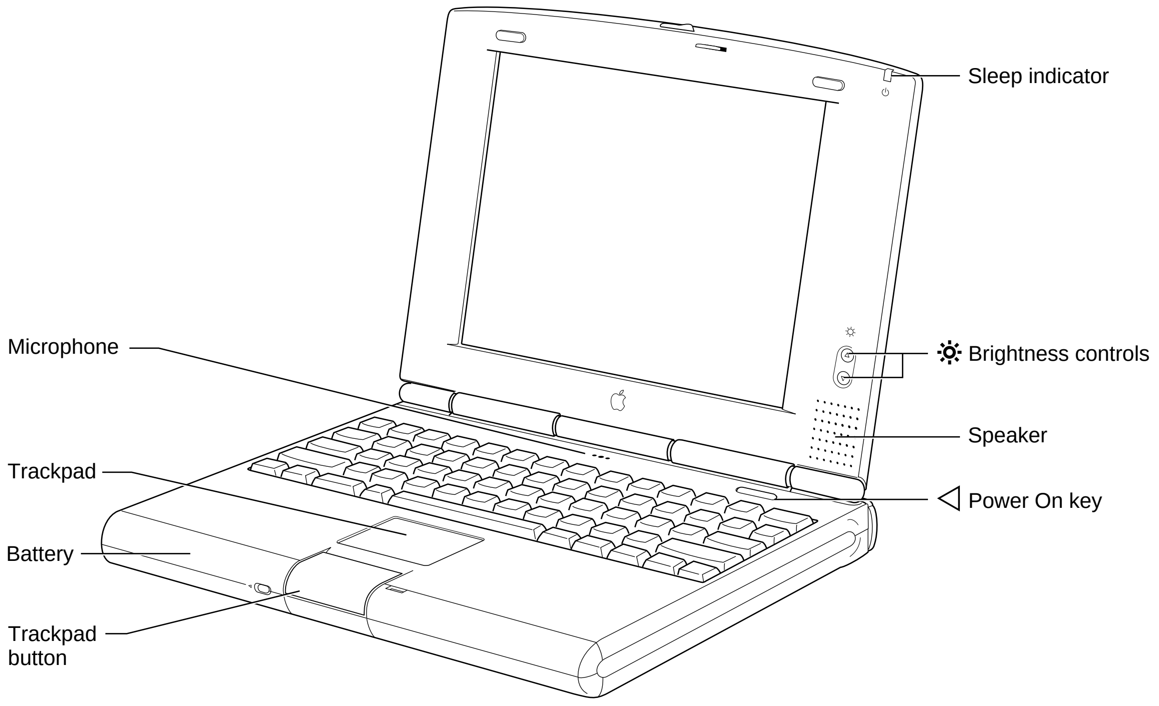
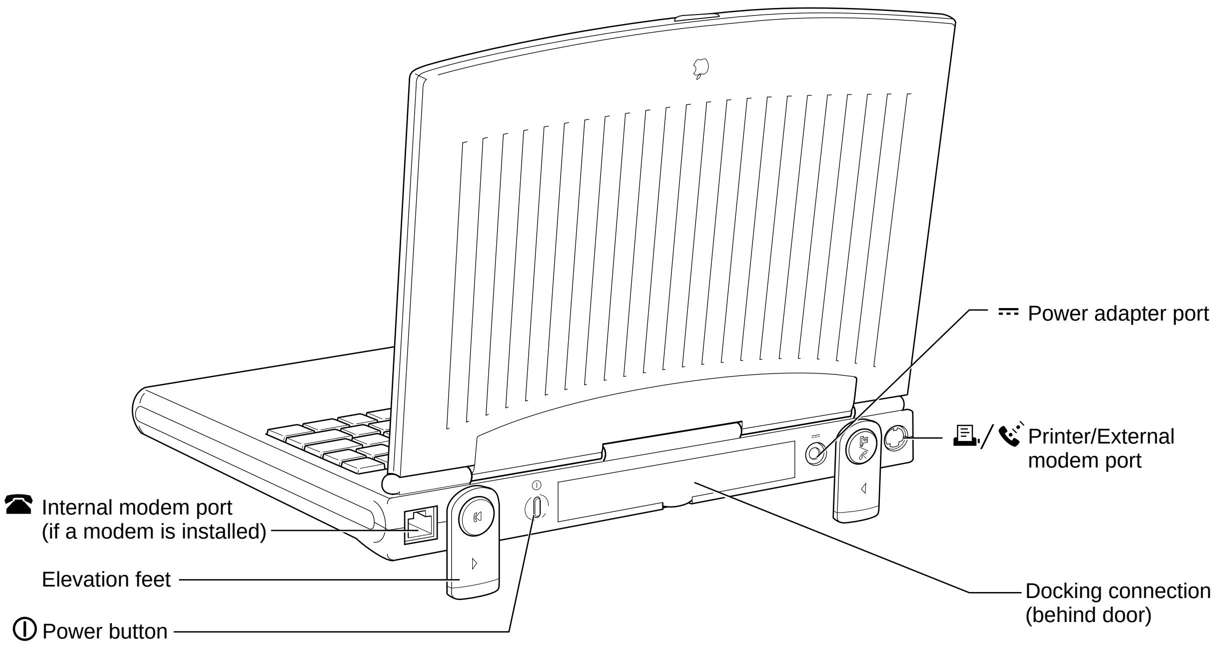


Figure 1-2 Back view of the computer


Power adapter port
Printer/External modem port
Internal modem port (if a modem is installed)
Elevation feet
Power button
Docking connection (behind door)

Accessory Devices

In addition to the devices that are included with the Macintosh PowerBook Duo 2300c computer, the following accessory devices are available:

- The PowerBook Duo 8 MB Memory Expansion Kit expands the RAM in the computers to 16 MB.
- The PowerBook Duo Battery Type III is available separately as an additional or replacement battery.
- The Power Adapter II, the AC adapter that comes with the computers, is also available separately.

Other accessories for the Macintosh PowerBook Duo family will work with the Macintosh PowerBook Duo 2300c computer, including memory expansion, modems, and the Duo Dock Plus.

Compatibility Issues

The Macintosh PowerBook Duo 2300c computer has several new features that distinguish it from the earlier models in the PowerBook Duo family. This section highlights key areas you should investigate to ensure that your hardware and software work properly with these new computers.

Size of Case

Because the clamshell case of the PowerBook Duo 2300c computer is slightly deeper than the case of the original PowerBook Duo, you cannot use the original Duo Dock with the PowerBook Duo 2300c. The slot in the Duo Dock is not deep enough to accommodate the computer's case. Users have two solutions: they may upgrade the top shell of an existing Duo Dock to make the slot deeper, or they may purchase the Duo Dock Plus, which can accommodate the deeper case without modification.

Microprocessor Differences

Differences between the PowerPC 603 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 603.

Completion Serialized Instructions

Completion serialized instructions cannot be executed until the execution of all prior instructions has been 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 603 has separate caches for instructions and data. Because the caches are separate, applications that mix code and data can encounter cache coherency problems.

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.

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. If you have situations such as these, 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.

Although it is essential to use 16-bit alignment for data 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 603.

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 603.

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 make sure their code is free of POWER instructions.

Power Manager Interface

Developers have written software that provides expanded Power Manager control for some older PowerBook models. That software will not work in the Macintosh PowerBook Duo 2300c 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 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 take care never to assume

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

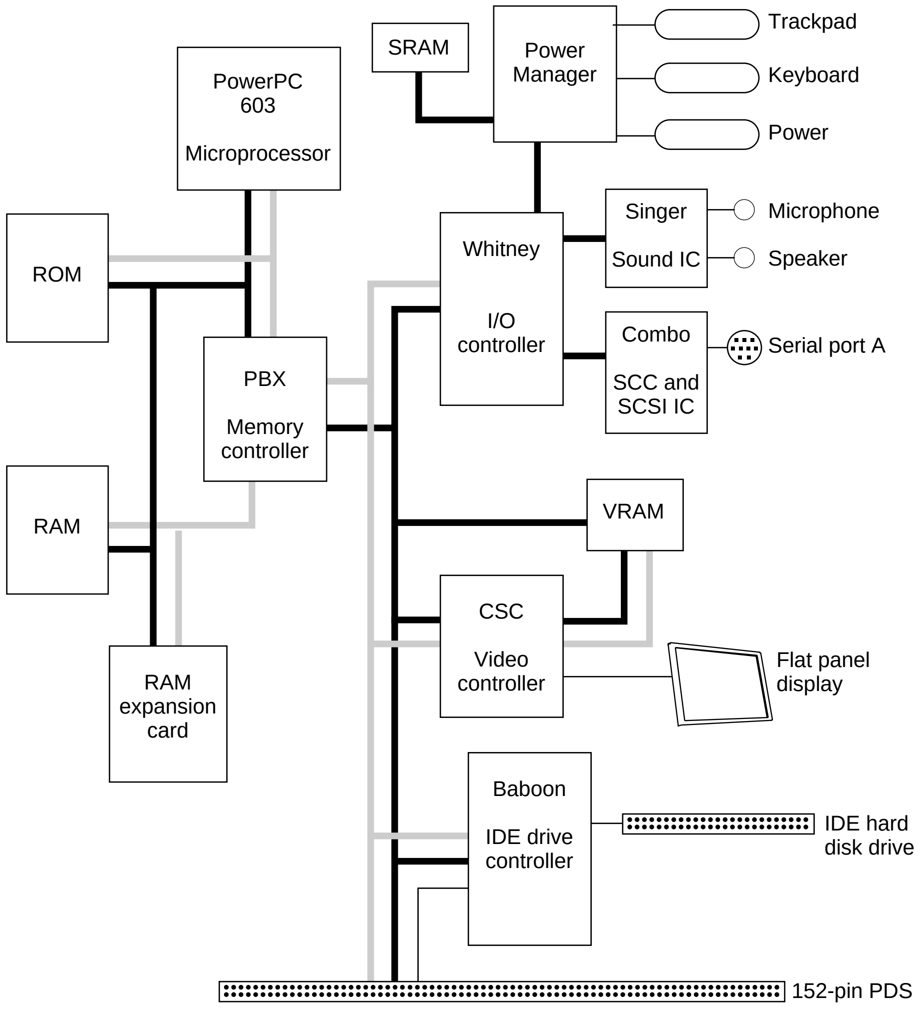
C H A P T E R 2

Architecture

The architecture of the Macintosh PowerBook Duo 2300c computer is partitioned into two subsystems: the processor and memory subsystem and the I/O subsystem. An Apple custom IC called the PBX IC acts as the bridge between the two subsystems.

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

Figure 2-1 Block diagram



Processor and Memory Subsystem

The processor and memory subsystem includes the PowerPC 603 microprocessor, the main RAM, the ROM, and the PBX memory controller IC. The processor and memory subsystem operates at 33 MHz on the PowerPC 603 bus. An optional RAM expansion card can be plugged into the computer and becomes part of this subsystem.

Main Processor

The main processor in the Macintosh PowerBook Duo 2300c computer is a PowerPC 603e microprocessor, an enhanced version of the PowerPC 603. Its principal features include

- full RISC processing architecture
- parallel processing units: one integer and one floating-point
- a load-and-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 (MMUs), one for instructions and one for data
- two separate on-chip caches of 16 KB each for data and instructions

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

RAM

The built-in RAM consists of 8 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 70-pin connector on the main logic board. With the RAM expansion card installed, the processor and memory subsystem supports up to 56 MB of RAM.

The RAM expansion card for the Macintosh PowerBook Duo 2300c computer is compatible with the one used in earlier PowerBook Duo models. The computer accepts up to 48 MB on a RAM expansion card.

The PBX custom IC contains bank base registers for making RAM banks contiguous, starting at address $0000 0000. See “PBX Memory Controller IC” on page 10.

ROM

The ROM in the Macintosh PowerBook Duo 2300c computer is implemented as a new array (1 M by 32-bit) consisting of two 1 M by 16-bit ROM ICs with an access time of 120 ns. These 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 is addressable only as longwords. See Chapter 4, “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 and the I/O bus.

Memory Control

The PBX IC controls the system RAM and ROM and provides address multiplexing and refresh signals for the DRAM devices.

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 an 68030 microprocessor. The other 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 clock rates.

I/O Subsystem

The I/O subsystem in the Macintosh PowerBook Duo 2300c computer operates at a clock frequency of 22 MHz on the I/O bus, a 68030-compatible bus. The I/O 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 IC
- the Power Manager IC

- the display controller IC
- the Baboon disk drive interface IC

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
- CPU ID register

The Whitney IC also performs the following functions:

- bus error timing for I/O bus
- bus arbitration for 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 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 an internal SCSI hard drive; it is needed only for upgrades to older PowerBook Duo models.

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.* Sound samples are transferred in or out through the Singer IC from sound I/O buffers maintained in main memory by the Whitney IC.

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

The CSC (color support chip) IC provides the data and control interface to the LCD panel. The CSC IC is also used in the Macintosh PowerBook 500 and 280 series computers. The CSC IC contains a 256-entry CLUT, RAMDAC, display buffer controller, and flat panel control circuitry. For more information, see “Displays” on page 14.

Baboon Disk Drive IC

The Baboon custom IC provides the interface to the IDE hard disk drive. For more information, see the section “Internal IDE Hard Disk Drive” beginning on page 15.

C H A P T E R 3

Input and Output Features

Input and Output Features

This chapter describes I/O features of the Macintosh PowerBook Duo 2300c computer, with emphasis on the features that have changed from earlier Macintosh PowerBook Duo computers.

IMPORTANT
The docking features of the Macintosh PowerBook Duo 2300c computer are the same as those of earlier PowerBook Duo models. For information about the docking connector, please refer to the *Macintosh PowerBook Duo Developer Note*. ▲

Displays

The Macintosh PowerBook Duo 2300c computer has a built-in color display. It is a liquid-crystal flat panel using active-matrix TFT (thin-film transistor) technology and has a built-in backlight using a CCFL (cold cathode fluorescent) lamp.

The active-matrix technology provides a high contrast ratio (60:1) and a response time of approximately 60 ms for performance similar to a CRT video display and with no cursor smearing or cursor submarining. The display normally displays black characters on a white background, simulating the appearance of a printed page.

Note
The color display in the Macintosh PowerBook Duo 2300c computer is 3.8 mm thicker than the grayscale display used in the original PowerBook Duo, requiring the Macintosh PowerBook Duo 2300c computer's case to be thicker than that of the original PowerBook Duo. ◆

The color display can operate in either of two modes. In 8-bit mode the display has a 640-by-480-pixel area and can display up to 256 different colors at a time. In 16-bit mode the display has a 640-by-400-pixel area and can display thousands of colors.

Smearing and Submarining

Older types of flat panel displays have much slower response times than the active-matrix displays used in the PowerBook Duo 280 and 280c. On those older displays, the pixels showing the cursor in one position do not clear quickly when the cursor moves to another position; as a result, the cursor appears smeared out in the direction of motion.

When the cursor is moving rapidly, the pixels may not have time to respond to a newly drawn cursor before the cursor moves to another position. In that case, the cursor seems to disappear behind the screen, an effect known as submarining.

The displays on the PowerBook Duo 280 and 280c do not have these anomalies.

The user can select either color display mode by using the Monitors control panel. Because the VRAM is a fixed size, 256K by 16 bits, it can handle only a certain amount of data. When the user selects 16-bit mode, the system software resizes the display area down to 400 lines instead of 480 and centers the display area on the screen, leaving black bands of 40 lines each at the top and bottom of the screen.

Note
The number of colors available in 16-bit mode is less than the theoretical maximum due to the limitations in the color LCD technology. Many color values exhibit noticeable flicker. The computer's CLUT omits the unsatisfactory colors, making about 4000 available. See the appendix, "Color Lookup Table," for more information. ◆

Internal IDE Hard Disk Drive

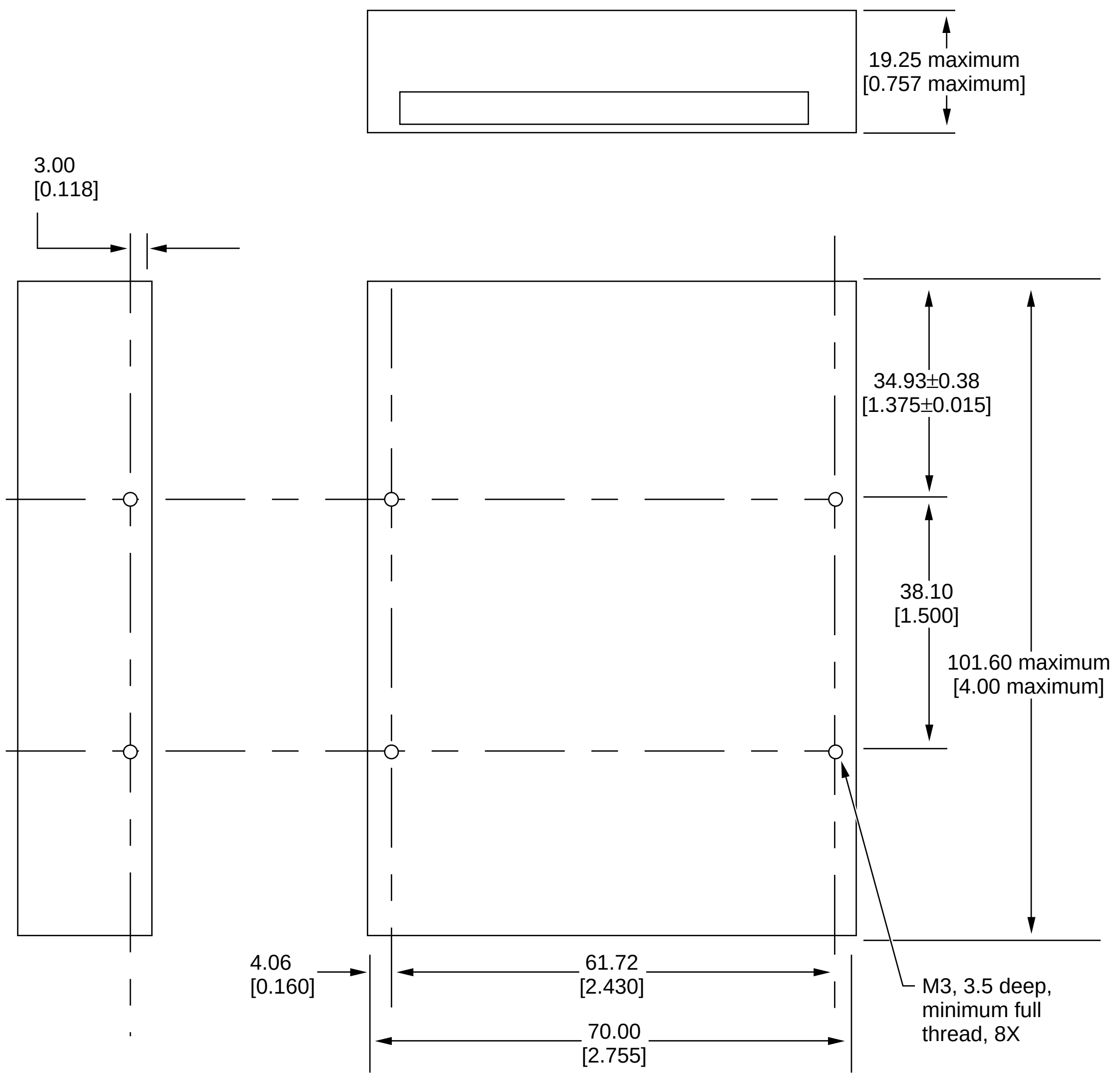
The Macintosh PowerBook Duo 2300c computer has an internal hard disk that uses the standard IDE interface. This interface, used for IDE drives on IBM AT–compatible computers, is also referred to as the ATA interface. The implementation of the ATA interface on the Macintosh PowerBook Duo 2300c computer is a subset of the ATA/IDE specification, ANSI proposal X3T10/0948D, Revision 2K (ATA-2).

For information about the software interface, see Chapter 7, "Software for the ATA Hard Disk."

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 conductive components and the bottom of the mounting envelope is 0.5 mm.

Figure 3-1 Maximum dimensions of the internal IDE hard disk

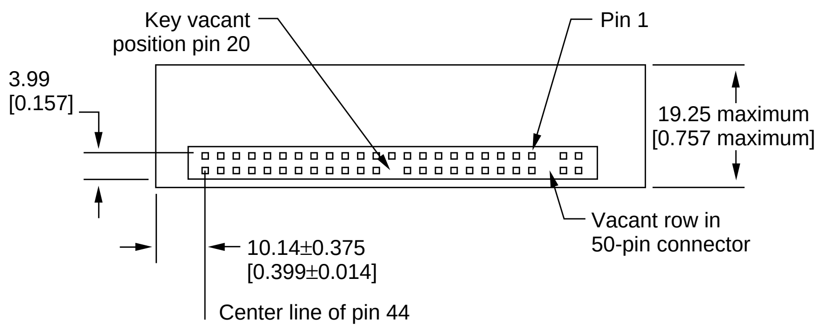


Note: Dimensions are in millimeters [inches].

Hard Disk Connector

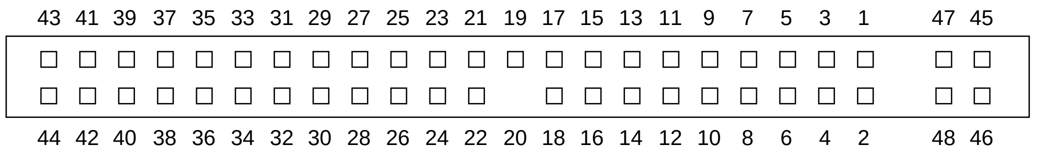
The internal hard disk has a 44-pin connector that carries both the IDE signals and the power for the drive. Figure 3-2 shows the location of the connector on the hard disk. Figure 3-3 identifies the pins. Pin 20 has been removed to serve as a key.

Figure 3-2 Location of the connector on the hard disk


Key vacant position 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]

Figure 3-3 Connector pin arrangement



Note: Gaps are equivalent to missing pins.

Pin Assignments

Table 3-1 shows the pin assignments on the 44-pin IDE 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
9	DD4	10	DD11
11	DD3	12	DD12
13	DD2	14	DD13

continued

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

Pin number	Signal name	Pin number	Signal name
15	DD1	16	DD14
17	DD0	18	DD15
19	GROUND	20	KEY
21	Reserved	22	GROUND
23	DIOW	24	GROUND
25	DIOR	26	GROUND
27	IORDY	28	Reserved
29	Reserved	30	GROUND
31	INTRQ	32	/IOCS16
33	DA1	34	Reserved
35	DA0	36	DA2
37	/CS0	38	/CS1
39	Reserved	40	GROUND
41	+5V	42	+5V
43	GROUND	44	Reserved

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

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.
/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.
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.
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.
/RESET	Hardware reset to the drive; an active-low signal.
Key	This pin is the key for the connector.

Terminator

The hard disk has 1000-ohm termination resistors for all I/O signal lines. The lines are pulled up through the resistors to the terminator power signal.

Power Requirements

Power drawn by the hard disk signals in each operating mode must be less than or equal to the values shown in Table 3-3. All measurements are under nominal environmental and voltage conditions. The limits include 1000-ohm pull-up resistors on all signal lines.

Table 3-3 Hard disk power requirements

	Current (amperes)	
Mode	Mean	Maximum
Startup*	—	1.30
Random operation†	0.50	0.60
Idle	0.30	0.35

* Startup values are peak values during response time of power on to power ready.
† Random operation values are RMS values with a 40 percent random seek, 40 percent write/read (1 write in 10 reads), and 20 percent idle mode.

C H A P T E R 4

Software Features

Software Features

This chapter describes the new features of the software for the Macintosh PowerBook Duo 2300c computer. The software includes the built-in ROM software and the system software that resides on the hard disk.

ROM Software

The ROM software in the Macintosh PowerBook Duo 2300c computer is based on the ROM used in previous Macintosh PowerBook computers, with enhancements to support the new features of this computer. 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
- IDE disk mode
- Ethernet
- trackpad

The following sections describe each of these features.

PowerPC 603 Microprocessor

The PowerPC 603 has power-saving modes similar to 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 Duo 2300c computer does not provide the Economode reduced-speed feature found on the 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 Duo 2300c computer is 118 (hexadecimal $76). *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 has been rewritten to accommodate

- 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 Duo 2300c computer uses a modified version of the public API for power management described in *Inside Macintosh: Devices*. See Chapter 5, "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.

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.

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 M2 computer, the internal hard disk is an IDE drive and the disk access mode is called IDE target mode.

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

Note
The ATA driver is described in Chapter 7,
"Software for the ATA Hard Disk." ◆

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.

Note
The Ethernet connector is provided on the Duo Dock. ◆

Trackpad Software

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 Macintosh Technical Note HW 01, *ADB—The Untold Story: Space Aliens Ate My Mouse.*

System Software

The Macintosh PowerBook Duo 2300c computer is shipped with new system software based on System 7.5 and augmented by several new features.

IMPORTANT
Even though the software for the PowerBook Duo 2300c 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 the IDE hard disk drive
- large partition support
- Drive Setup, a new utility
- improved file sharing
- a new Dynamic Recompilation Emulator
- a Resource Manager completely in native code
- an improved math library
- new `BlockMove` extensions
- 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 in later chapters. Please see the cross-references in the individual sections that follow. ◆

Control Strip

The desktop on the Macintosh PowerBook Duo 2300c computer includes the status and control element called the control strip, which was introduced in the Macintosh PowerBook 280 and 500 models. It is a strip of graphics with small button controls and indicators in the form of various icons. For a description of the control strip and guidelines for adding modules to it, see Macintosh Technical Note OS 6, *Control Strip Modules,* on the reference library edition of the developer CD.

Support for IDE Disk Drives

Support for IDE (integrated drive electronics) hard disk drives is incorporated in the ROM software. System software for controlling IDE hard drives is included in a new ATA device driver and the ATA Manager. The new driver and manager are described in Chapter 7, "Software for the ATA Hard Disk."

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 data types, please see "The API Modifications" in Chapter 6, "Large Volume Support."

System-Level Software

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

- virtual memory code
- Disk Init
- FSM Init
- 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 Folder and Get Info windows and to 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 the Finder's.

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 Duo 2300c computer includes a new disk setup utility named Drive Setup that replaces the old HDSC Setup utility. In addition to the ability to support large volumes, the Drive Setup utility has several other enhancements, including

- an improved user interface
- support for multiple partitions
- support for chainable drivers

- support for multiple HFS partitions
- the ability to mount volumes from within the Drive Setup application
- 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.

Dynamic Recompilation Emulator

The Dynamic Recompilation Emulator (or DR Emulator) is an enhancement of the current interpretive emulator. It provides on-the-fly translation of 680x0 instructions into PowerPC instructions with improved performance compared with the current 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.

Although the DR Emulator provides a high degree compatibility for 680x0 code, it is less compatible than that of the current emulator 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 and should be avoided. See also “Split Cache” on page 5.

Resource Manager in Native Code

The Resource Manager in the software for the Macintosh PowerBook Duo 2300c computer is similar to the one in the first Power Macintosh computers except that it is completely in native PowerPC code. Because the Resource Manager is intensively used both by system software and by 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.

MathLib 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.

MathLib has been improved to better exploit the floating-point features of the PowerPC microprocessor. MathLib 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, MathLib improves performance without compromising accuracy or robustness. For often-used functions, it provides performance gains 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 Duo 2300c 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 604 processors, rather than for 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 0.

Note
The new `BlockMove` extensions do not use 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 4-1.

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 4-1 Summary of `BlockMove` routines

`BlockMove` version	Can be called from 680x0 code	OK to use for moving 680x0 code	OK 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

Table 4-1 summarizes the `BlockMove` routines according to three criteria: whether the routine can be called from 680x0 code, whether it is OK to use for moving 680x0 code, and whether it is OK to use with buffers or other uncacheable destination locations.

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 prefetch 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

Because the PowerPC 603 microprocessor used in the Macintosh PowerBook Duo 2300c computer does not support the POWER-only instructions, 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 604 microprocessor. ◆

Here is a list of the POWER-clean native code fragments in the system software for the Macintosh PowerBook Duo 2300c 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

The first 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 Duo 2300c computer, the emulation code has been expanded 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 Power Macintosh computers that use the PowerPC 601. ◆

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 Duo 2300c 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 Duo 2300c 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 Power PC version of native QuickDraw. It allows a patch chaining mechanism for decisions on categories of block-transfer 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 5

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 Duo 2300c 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 Duo 2300c computer and for future Macintosh PowerBook models includes an interface for program access to the Power Manager's functions, so it is no longer necessary for applications to deal directly with the Power Manager's data structures. The functions 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's interface functions, it's always a good idea to call the Gestalt Manager to see if the the routines are 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 functions may be added in the future, one of the new functions simply returns the number of functions that are implemented. The following code fragment determines both that the routines in general exist and that at least the hard disk spindown function 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 function that is not implemented, 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 Functions

This section tells you how to call the interface functions for the Power Manager software. The interface functions are listed here in the order of their selector values, as shown in Table 5-1.

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

Table 5-1 Interface functions and their selector values

	Selector value	
Function 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

PMSelectorCount

You can use the `PMSelectorCount` function to determine which functions are implemented.

```
short PMSelectorCount();
```

DESCRIPTION

The `PMSelectorCount` function returns the number of function selectors present. Any function 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` function to find out which features of the Power Manager are implemented.

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

DESCRIPTION

The `PMFeatures` function 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 wake-up 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.

continued

Bit name	Bit number	Description
`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` function to find out how long the computer will wait before going to sleep.

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

DESCRIPTION

The `GetSleepTimeout` function 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 time-out value is returned in the low word of register D0.

SetSleepTimeout

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

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

DESCRIPTION

The `SetSleepTimeout` function sets the amount of time the computer will wait after the last user activity before going to sleep. The value of `timeout` is expressed as the number of 15-second intervals that make up the desired time. If a value of 0 is passed in, the function sets the time-out 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 time-out value to set is passed in the high word of register D0.

GetHardDiskTimeout

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

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

DESCRIPTION

The `GetHardDiskTimeout` function 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 time-out value is returned in the low word of register D0.

SetHardDiskTimeout

You can use the `SetHardDiskTimeout` function 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` function 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 function 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` function to find out whether the internal hard disk is on.

```
Boolean HardDiskPowered();
```

DESCRIPTION

The `HardDiskPowered` function 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` function to force the hard disk to spin down.

```
void SpinDownHardDisk();
```

DESCRIPTION

The `SpinDownHardDisk` function 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` function to find out whether hard disk spindown is enabled.

```
Boolean IsSpindownDisabled();
```

DESCRIPTION

The `IsSpindownDisabled` function returns the Boolean value `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` function to disable hard disk spindown.

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

DESCRIPTION

The `SetSpindownDisable` function 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` function, 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` function to notify your software when power to the internal hard disk is about to be turned off.

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

DESCRIPTION

The `HardDiskQInstall` function 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 function pointed to by the `hdProc` field so that it can do any special processing. The software passes the function a pointer to its queue element so that, for example, the function 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` function 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` function 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` function to find out the condition of the battery or batteries.

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

DESCRIPTION

The `GetScaledBatteryInfo` function provides a generic means of returning information about the battery or batteries in the system. Instead of returning a voltage value, the function 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 battery voltage that is 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` function 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 function 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` function to turn the automatic sleep feature on and off.

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

DESCRIPTION

The `AutoSleepControl` function 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` multiple times with `enableSleep` set to `false` 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` function to find out information about the internal modem.

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

DESCRIPTION

The `GetIntModemInfo` function 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.

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`).

continued

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` function to set some parts of the state of the internal modem.

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

DESCRIPTION

The `SetIntModemState` function 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` function to find out the maximum speed of the computer's microprocessor.

```
short MaximumProcessorSpeed();
```

DESCRIPTION

The `MaximumProcessorSpeed` function 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` function to find out the current clock speed of the microprocessor.

```
short CurrentProcessorSpeed();
```

DESCRIPTION

The `CurrentProcessorSpeed` function 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` function to find out whether the computer will run at full speed the next time it restarts.

```
Boolean FullProcessorSpeed();
```

DESCRIPTION

The `FullProcessorSpeed` function 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` function to set the clock speed the microprocessor will use the next time the computer is restarted.

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

DESCRIPTION

The `SetProcessorSpeed` function 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 50). 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` function to find out the SCSI ID the computer uses in SCSI disk mode.

```
short GetSCSIDiskModeAddress();
```

DESCRIPTION

The `GetSCSIDiskModeAddress` function 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.

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` function to set the SCSI ID for the computer to use in SCSI disk mode.

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

DESCRIPTION

The `SetSCSIDiskModeAddress` function 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` function to find out when the computer will wake up from sleep mode.

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

DESCRIPTION

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

If the PowerBook model doesn't support the wake-up 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;      /* wake-up 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` function to set the time when the computer will wake up from sleep mode.

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

DESCRIPTION

The `SetWakeupTimer` function 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` function to find out whether processor cycling is enabled.

```
Boolean IsProcessorCyclingEnabled();
```

DESCRIPTION

The `IsProcessorCyclingEnabled` function 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` function to turn the processor cycling feature on and off.

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

DESCRIPTION

The `EnableProcessorCycling` function 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` function to find out how many batteries the computer supports.

```
short BatteryCount();
```

DESCRIPTION

The `BatteryCount` function 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` function to find out the battery voltage.

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

DESCRIPTION

The `GetBatteryVoltage` function 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` function to find out about how much battery time remains.

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

DESCRIPTION

The `GetBatteryTimes` function 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 6

Large Volume Support

This chapter describes the large volume file system for the Macintosh PowerBook Duo 2300c 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 Duo 2300c 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 extended `PBXGetVolInfo` function are described in “The API Modifications” beginning on page 67.

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 previously were 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 and time of creation
   unsigned long     ioVLsMod;        // date and time of last modification
   unsigned short    ioVAtrb;         // volume attributes
   unsigned short    ioVNmFls;        // number of files in root directory
   unsigned short    ioVBitMap;       // first block of volume bitmap
   unsigned short    ioAllocPtr;      // first block of next new file
   unsigned short    ioVNmAlBlks;     // number of allocation blocks
   unsigned long     ioVAlBlkSiz;     // size of allocation blocks
   unsigned long     ioVClpSiz;       // default clump size
   unsigned short    ioAlBlSt;        // first block in volume map
   unsigned long     ioVNxtCNID;      // next unused node ID
   unsigned short    ioVFrBlk;        // number of free allocation blocks
   unsigned short    ioVSigWord;      // volume signature
   short             ioVDrvInfo;      // drive number
   short             ioVDRefNum;      // driver reference number
   short             ioVFSID;         // file-system identifier
   unsigned long     ioVBkUp;         // date and 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.

`ioWPosOffset`	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.

continued

←	`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 `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

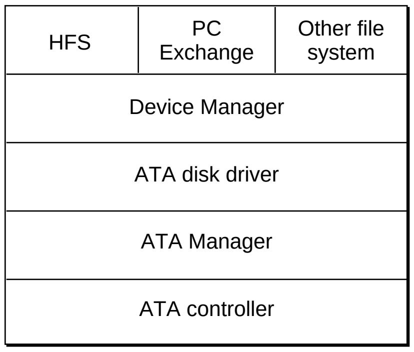
Software for the ATA Hard Disk

This chapter describes the system software that controls ATA (IDE) hard disk drive in the Macintosh PowerBook Duo 2300c 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

The ATA software in the Macintosh PowerBook Duo 2300c computer conforms to the Macintosh driver model. File systems communicate with the driver by way of the Device Manager, as shown in Figure 7-1. The ATA software consists of the ATA Manager and the ATA disk driver.

Figure 7-1 ATA software model



At the system level, the ATA disk driver and the ATA Manager work in the same way that the SCSI Manager and associated SCSI device drivers work. The ATA disk driver provides drive partition, data management, and error-handling services for the operating system as well as support for determining device capacity and controlling 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 Duo 2300c computer has the following features:

- supports all ATA drives that comply with the ANSI ATA/IDE 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 Macintosh PC Exchange for DOS file compatibility.
- adheres to the driver rules described in *Designing PCI Cards and Drivers for Power Macintosh computers*
- supports both synchronous and asynchronous requests from the file system.

The ATA disk driver resides in ROM and supports all ATA drives that adhere to the ANSI ATA/IDE specification X3T10.

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 78.

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.

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.

The functions and data structures of the ATA Manager are described in “ATA Manager Reference” beginning on page 93.

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 80 and “Status Functions” beginning on page 88.

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

The Control Routine

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

The ATA disk driver implements many of the control functions supported by the SCSI hard disk device driver and defined in *Inside Macintosh: Devices* plus several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh computers.* The control functions are listed in Table 7-1 and described in “Control Functions” beginning on page 80.

Table 7-1 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
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,* plus several new ones that are defined in *Designing PCI Cards and Drivers for Power Macintosh computers*. The status functions are listed in Table 7-2 on page 80 and described in "Status Functions" beginning on page 88.

Table 7-2 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 Macintosh PC Exchange. It requests the driver to redefine the starting block offset and capacity of an existing partition.

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

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

Parameter block

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

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

Parameter block

→	`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, and sleep. It can be used to reduce drive power consumption and decrease system noise levels.

IMPORTANT

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

- Active: The fully operational state with typical power consumption.
- Standby: The state with minimal power savings. The device can return to the active state in less than 5 seconds.

- Idle: The state with moderate power savings. The device can return to the active state within 15 seconds.
- Sleep: The state with minimum power consumption. The device can return to the active state within 30 seconds.

Parameter block

→	`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 Gestalt parameter block:

```
struct DriverGestaltParam
{
   ataPBHdr                                  // See definition on page 93
   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'`	Specifies type of device the driver is controlling.
`'intf'`	Specifies the device interface.
`'boot'`	Specifies PRAM value to designate this driver or device.
`'vers'`	Specifies the version number of the driver.
`'lpwr'`	Indicates support for low-power mode.
`'dAPI'`	Indicates support for Macintosh PC Exchange calls.
`'purg'`	Indicates driver can be closed or purged.
`'wide'`	Indicates large volume support.
`'ejec'`	Eject-call requirements.

←	`driverGestaltResponse`	Returned result 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 and 0 if not enabled or if the partition does not have a partition map entry on the media. The partition is specified either by its associate logical drive or the partition's block address on the media.

Parameter block

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

RESULT CODES

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

get partition write protect status

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

Parameter block

→	`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 Macintosh PC Exchange. It requests the driver to return information about the partition specified by `ioVRefNum`.

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

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

Parameter block

→	`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 87.

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

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

An arrow preceding the comment indicates whether the parameter is an input parameter, an output parameter, or both.

Arrow	**Meaning**
→	Input
←	Output
↔	Both

The ATA parameter block header structure is defined as follows:

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

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 nonzero value indicates an asynchronous I/O request. The routine this field points to is called when the request has finished without error or when the request

has terminated due to an error. This field is valid for any manager request. The completion routine is called as follows:

```
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 7-13 on page 134 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 7-4 on page 99. 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 drive identification data. Currently values 0 through 3 are supported, as defined in the ATA/IDE specification. See the ATA/IDE specification for the definitions of the timing values. If a timing value higher than one supported is specified, the manager operates in the fastest timing mode supported by the manager. Until the timing value is determined by examining the drive identification data returned by the `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 7-3 on page 96.

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

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

Name	Bit	Definition
`LED Enable`	0	Some systems are equipped with an activity LED controlled by software. Setting this bit to 1 indicates that the LED should be turned on for this transaction. The LED is automatically turned off at the end of the transaction. Setting the bit to 0 indicates that the LED should not be turned on for this transaction. This bit has no effect in systems with no activity LED.
—	1–2	Reserved.
`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 107 for details. The following device registers are reported back: Sector count register Sector number register Cylinder register(s) SDH register
`ProtocolType`	4–5	These 2 bits specify the type of command. The following command types are defined: $0 = standard ATA $1 = reserved $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.

continued

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

Name	Bit	Definition

`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;`

`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.

continued

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

Name	Bit	Definition
`Immediate`	11	When this bit is set to 1, it indicates that the request must be executed as soon as possible and that the status of the request must be returned. It forces the request to the head of the I/O queue for immediate execution. When this bit is set to 0, the request is queued in the order it is received and is executed according to that order.
`ATAioDirection`	12, 13	This bit field specifies the direction of data transfer. Bit values are binary and are 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 7-4.

Table 7-4 ATA Manager functions

Function name	Code	Description
`ATA_Abort`	$10	Terminate the command.
`ATA_BusInquiry`	$03	Get bus information.
`ATA_DrvrRegister`	$85	Register the driver reference number.
`ATA_DrvrDeregister`	$87	Deregister the driver reference number.
`ATA_EjectDrive`	$89	Auto-eject the drive.
`ATA_ExecIO`	$01	Execute ATA I/O.
`ATA_FindRefNum`	$86	Look up the driver reference number.
`ATA_GetDevConfig`	$8A	Get the device configuration.
`ATA_GetDevLocationIcon`	$8C	Get the device location icon and string.
`ATA_Identify`	$13	Get the drive identification data.
`ATA_MgrInquiry`	$90	Get information about the ATA Manager and the system configuration.
`ATA_ModifyDrvrEventMask`	$88	Modify the driver event mask.
`ATA_NOP`	$00	Perform no operation.
`ATA_QRelease`	$04	Release the I/O queue.
`ATA_RegAccess`	$12	Obtain access to an ATA device register.
`ATA_ResetBus`	$11	Reset the ATA bus.
`ATA_SetDevConfig`	$8B	Set the device configuration.

ATA_Abort

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

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

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 94
   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 94.
`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 `ATA_BusInquiry` function is $03.

The parameter block associated with this function is defined below:

```
struct ATA_BusInquiry               // ATA bus inquiry structure
{
   ataPBHdr                         // see definition on page 94
   UInt16   ataEngineCount;         // ← TBD; 0 for now
   UInt16   ataReserved;            // reserved
   UInt32   ataDataTypes;           // ← TBD; 0 for now
   UInt16   ataIOpbSize;            // ← size of ATA I/O PB
   UInt16   ataMaxIOpbSize;         // ← TBD; 0 for now
   UInt32   ataFeatureFlags;        // ← TBD
   UInt8    ataVersionNum;       // ← HBA Version number
   UInt8    ataHBAInquiry;       // ← TBD; 0 for now
   UInt16   ataReserved2;        // reserved
   UInt32   ataHBAPrivPtr;       // ← pointer to HBA private data
   UInt32   ataHBAPrivSize;      // ← size of HBA private data
   UInt32   ataAsyncFlags;       // ← capability for callback
   UInt32   ataReserved3[4];     // reserved
   UInt32   ataReserved4;        // reserved
   SInt8    ataReserved5[16];    // TBD
   SInt8    ataHBAVendor[16];    // ← HBA Vendor ID
   SInt8    ataContrlFamily[16]; // ← family of ATA controller
   SInt8    ataContrlType[16];   // ← controller model number
   SInt8    ataXPTversion[4];    // ← version number of XPT
   SInt8    ataReserved6[4];     // reserved
   SInt8    ataHBAversion[4];    // ← version number of HBA
   UInt8    ataHBAslotType;      // ← type of slot
   UInt8    ataHBAslotNum;       // ← slot number of the HBA
   UInt16   ataReserved7;        // reserved
   UInt32   ataReserved8;        // reserved
};
typedef struct ATA_BusInquiry ATA_BusInquiry;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` on page 94.
`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 noninterrupt 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 Table 7-5 on page 104 for possible events.

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

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

Version two allows a driver to register as a Notify-all driver. Registration of a Notify-all driver is signaled by a value of –1 in the `deviceID` field of the header and 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 ensure proper operation, all PCMCIA / ATA and Notify all device drivers must register using version two, which provides event handling capability. ◆

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

```
// Version 1 (ataPBVers = 1)
struct      ataDrvrRegister   // parameter block structure
                              // for ataPBVers = 1
{
   ataPBHdr                   // header information
   SInt16   drvrRefNum;       // → driver reference number
   UInt16   FlagReserved;     // reserved -> should be 0
   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 0
   ProcPtr ataEHandlerPtr     // → event handler routine pointer
   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 94.
`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

```
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 127 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 into `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 be called only 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 7-5 lists the masks have been defined.

Table 7-5 Event masks

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

RESULT CODES

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

ATA_DrvrDeregister

You can use the `ATA_DrvrDeRegister` function to deregister the selected drive. After successful completion of this function, the driver reference number for the drive is set to 0, indicating that 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
Notify-all device drivers must deregister using the parameter version 2. ▲

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 94.
`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 to ensure 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 make sure that all partitions associated with the device have been dismounted from the desktop.

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

The data structure of the function is as follows:

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

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 94.
`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 94
   SInt8    ataStatusReg;  // ← last device status register image
   SInt8    ataErrorReg;   // ← last device error register
                           // (valid if bit 0 of Status field set)
   SInt16   ataReserved;   // reserved
   UInt32   BlindTxSize;   // → data transfer size
   UInt8*   ioBuffer;      // ↔ data buffer ptr
   UInt32   ataActualTxCnt;// ← actual number of bytes
                           // transferred
   UInt32   ataReserved2;  // reserved
   devicePB RegBlock;      // → device register images
   UInt8*   packetCDBPtr;  // ATAPI packet command block pointer
   UInt16   ataReserved3[6];// Reserved
};
typedef struct ATA_ExecIO ATA_ExecIO;
```

Field descriptions

`ataPBHdr`	See the parameter block definition on page 94.
`ataStatusReg`	This field contains the last device status register image. See the ATA/IDE specification for status register bit definitions.
`ataErrorReg`	This field contains the last device error register image. This field is valid only if the error bit (bit 0) of the `Status` register is set. See the ATA/IDE specification for error register bit definitions.
`ataReserved`	Reserved. All reserved fields are set to 0 for future compatibility.
`BlindTxSize`	This field specifies the maximum number of bytes that can be transferred for each interrupt or detection of a data request. Bytes are transferred in blind mode (no byte-level handshake). Once an interrupt or a data request condition is detected, the ATA Manager transfers up to the number of bytes specified in the field from or to the selected device. The typical number is 512 bytes.
`ioBuffer`	This field contains the host buffer address for the number of bytes specified in the `ioReqCount` field. Upon returning, this field is updated to reflect data transfers. When the `SGType` bits of the `ataFlags` field are set, this field points to a scatter gather list. The scatter gather list consists of series of `IOBlk` entries defined as follows:

```
struct IOBlk
{
   UInt8*   ioBuffer;   // ↔ data buffer ptr
   UInt32   ioReqCount; // ↔ transfer length
};
typedef struct IOBlk IOBlk;
```

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

calling the ATA Manager. The ATA device register image structure is defined as follows:

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

For ATAPI commands, the cylinder image must contain the preferred PIO DRQ packet size to be writtern 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 to ensure future compatibility. The packet structure for the ATAPI command is defined as follows:

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

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

RESULT CODES

`noErr`	Successful completion; no error occurred
`nsDrvErr`	Specified logical drive number does not exist
`AT_AbortErr`	Command aborted bit set in error register
`AT_RecalErr`	Track 0 not found bit set in error register
`AT_WrFltErr`	Write fault bit set in status register
`AT_SeekErr`	Seek complete bit not set upon completion

`AT_UncDataErr`	Uncorrected data bit set in error register
`AT_CorDataErr`	Data corrected bit set in status register
`AT_BadBlkErr`	Bad block bit set in error register
`AT_DMarkErr`	Data mark not found bit set in error register
`AT_IDNFErr`	ID-not-found bit set in error register
`ATABusy`	Selected device busy (BUSY bit set)
`ATAMgrNotInitialized`	ATA Manager not initialized
`ATAPBInvalid`	Invalid device base address detected (= 0)
`ATAQLocked`	I/O queue locked—cannot proceed
`ATAReqInProg`	I/O channel in use—cannot proceed
`ATATransTimeOut`	Timeout: transaction time-out detected
`ATAUnknownState`	Device in unknown state

ATA_FindRefNum

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

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

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

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

```
// Version 1 (ataPBVers = 1)
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 0
   ProcPtr  ataEHandlerPtr    // ← event handler routine pointer
   SInt32   drvrContext;      // ← 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 94.
`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_GetDevConfig

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

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

The data structure for the function is as follows:

```
struct      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 or I/O
   UInt8    pcVcc;            // ↔ Vcc voltage in tenths
   UInt8    pcVpp1;           // ↔ Vpp 1 voltage in tenths
   UInt8    pcVpp2;           // ↔ Vpp 2 voltage in tenths
   UInt8    pcStatus;         // ↔ Card Status register setting
   UInt8    pcPin;            // ↔ Card Pin register setting
   UInt8    pcCopy;           // ↔ Card Socket/Copy register
                              // setting
   UInt8    pcConfigIndex;    // ↔ Card Option register setting
   UInt16   Reserved[10];     // reserved
};
typedef struct ataGetDevConfiguration ataGetDevConfiguration;
```

Field descriptions

`ataPBHdr`	See the `ataPBHdr` parameter block definition on page 94.
`ConfigSetting`	This field indicates various configuration settings. The following bits have been defined: Bits 5–0: Reserved for future expansion (set to 0) Bit 6: ATAPI packet DRQ handling setting (only applies to ATAPI) 1: The function waits for an interrupt to happen before sending the ATAPI command packet.

	0: The function waits for the assertion of DRQ bit in the status register before sending the ATAPI command packet. This is the default setting.
	Bits 7–31: Reserved (set to 0)
`ataIOSpeedMode`	This field is reserved for future expansion.
`pcValid`	This field indicates which of the PCMCIA unique fields contain valid values. Table 7-6 on page 114 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 7-6 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 0.

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

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

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

```
struct DrvLocationIcon
{
   ataPBHdr                  // see above definition
   SInt16   ataIconType;     // → icon type specifier
   SInt16   ataIconReserved; // reserved; set to 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 94.
`ataIconType`	This field defines the type of icon desired as follows: $01 - large black-and-white icon with mask $81 - same as 1, but ProDOS icon
`ataIconReserved`	Reserved to be longword aligned. This field should be set to 0 for future compatibility.
`ataLocationIconPtr`	A pointer to the location icon buffer. When the pointer is nonzero, the function copies the icon data to the buffer.
`ataLocationStringPtr`	A pointer to the location string buffer. When the pointer is nonzero, 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 94
   SInt8    ataStatusReg;     // ← last ATA status image
   SInt8    ataErrorReg;      // ← last ATA error image;
                              // valid if error bit set
   SInt16   ataReserved;      // reserved
   UInt32   BlindTxSize;      // ← this field is set to 512
                              // upon returning
   UInt8    *DataBuf;         // buffer for the identify data
                              // (512 bytes)
   UInt32   ataRequestCount;  // ← indicates remaining
                              // byte count
   UInt32   ataActualTxCnt;   // ← actual transfer count
   UInt32   ataReserved2;     // reserved
   devicePB RegBlock;         // ← task file image sent for
                              // the command
   UInt16   Reserved3[8];     // used internally by ATA Manager
};
typedef struct ataIdentify ataIdentify;
```

Field descriptions

ataPBHdr	See the definition of the ataPBHdr parameter block on page 94.
ataStatusReg	Status register image for the last ATA task file.
ataErrorReg	Error register image last ATA task file . This field is only valid if the LSB (error bit) of the ataStatusReg field is set.
BlindTxSize	Byte size of the Identify data.
DataBuf	Pointer to the data buffer for the device identify data. The length of the buffer must be at least 512 bytes.
ataReserved, ataReserved2, Reserved3[8]	These fields are reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

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

ATA_MgrInquiry

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

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

The parameter block associated with this function is defined below:

```
struct ATA_MgrInquiry        // ATA inquiry structure
{
   ataPBHdr                  // see definition on page 94
   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 94.
`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 0 if the ATA Manager has not been initialized.
`ataDevCnt`	Upon return, this field contains the total number of ATA devices detected on all ATA buses. The current architecture allows only one device per bus. This field will contain 0 if the ATA Manager has not been initialized.
`ataMaxMode`	This field specifies the maximum I/O speed mode that the ATA Manager supports. Refer to the ATA/IDE specification for information on mode timing.

`IOClkResolution`	This field contains the I/O clock resolution in nanoseconds. The current implementation doesn't support the field (returns 0).
`Reserved[17]`	This field is reserved. To ensure future compatibility, all reserved fields should be set to 0.

RESULT CODES

`noErr`	0	Successful completion; no error occurred

ATA_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 nonregistered bus has no effect.

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

The manager function code for the `ATA_ModifyDrvrEventMask` function is $88.

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 94.
`modifiedEventMask`	New event mask setting. The definitions of the subfields are given in Table 7-5 on page 104.
`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. It returns `noErr` if the drive number is valid.

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

The parameter block associated with this function is defined below:

```
lstruct     ataNOP                 // parameter block structure
{
   ataPBHdr                        // see definition on page 94
   UInt16   Reserved[24];          // reserved
};
typedef struct ataNOP ataNOP;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` on page 94.

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 94
   UInt16   Reserved[24];         // Reserved
};
typedef struct ataQRelease ataQRelease;
```

Field descriptions

`ataPBHdr`	See the definition of `ataPBHdr` on page 94.

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 94
   UInt16   RegSelect;            // → Device Register selector
   union    {
            UInt8  byteRegValue;  // ↔ byte register value read
                                  // or to be written
            UInt16 wordRegValue;  // ↔ word register value read
                                  // or to be written
   } registerValue;
   UInt16   Reserved[22];         // reserved
};
typedef struct ataRegAccess ataRegAccess;
```

```
// Version 2 (ataPBVers = 2)
struct      ataRegAccess        // parameter block structure
                                // for ataPBVers of 2
{
   ataPBHdr                     // see definition on page 94
   UInt16   RegSelect;          // → Device Register selector
   union    {
            UInt8  byteRegValue; // ↔ register value read or
                                // to be written
            UInt16 wordRegValue; // ↔ word register value read
                                // or to be written
   } registerValue;
   // The following fields are valid only if RegSelect = $FFFF
   UInt16   regMask;            // → mask indicating which
                                // combination of registers
                                // to access.
   devicePB ri;                 // ↔ register images
                                // (Feature - Command)
   UInt8    altStatDevCntrReg;  // ↔ Alternate Status (R) or
                                // Device Control (W) register
   UInt8    Reserved3;          // reserved (set to 0)
   UInt16   Reserved[16];       // reserved
};
typedef struct ataRegAccess ataRegAccess;
```

Field descriptions

`ataPBHdr`	See the definition of the `ataPBHdr` parameter block on page 94.
`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 7-7.
`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 `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 7-8.

`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 the section "ATA_ExecIO" on page 107 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 1.

Table 7-7 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 7-8 Register mask bits

Bit number	Definition
0	Data register
1	Error register (R) or Feature register (W)
2	Sector Count register
3	Sector Number register
4	Cylinder Low register
5	Cylinder High register
6	SHD register

continued

Table 7-8 Register mask bits (continued)

Bit number	Definition
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 94
   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 94.
`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 manager function code for the `ATA_SetDevConfig` function is $8B.

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 94.
`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 (applies only to ATAPI) 1 = The function waits for an interrupt to happen before sending the ATAPI command packet. 0 = The function waits for the assertion of DRQ bit in the status register before sending the ATAPI command packet. This is the default setting. Bits 7–31: Reserved (set to 0)
`ataIOSpeedMode`	This field is reserved for future expansion.
`pcValid`	This field indicates which of the PCMCIA unique fields contain valid values. Table 7-6 on page 114 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 the asynchronous event-reporting mechanism of the Card Services Manager, the ATA Manager notifies its clients by a callback mechanism using the client's event handler. Each client that is to be notified of device events must register its event handler at the time of driver registration. Refer to the section "ATA_DrvrRegister" beginning on page 102 for the calling convention of the event handler.

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

Table 7-9 Event codes send by the ATA Manager

Event code	Event description
$00	Null event; signifies no real event. The client should simply return with no error code.
$01	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 repowered from sleep/low power. The client should let the operating system know about the presence of the device (if it has not done so already), verify the device type, and upload any device characteristics.
$02	Offline event; signifies that the device has gone offline. This event may happen as a result of a device being manually removed from the socket. 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 operating system 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 130.

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

Table 7-10 Input parameter bits for the old API

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

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

Table 7-11 Input parameter bits for the new API

Bits	Value	Definition	
31–24	Reserved	In this byte, bits 29–31 are currently defined. All other bits should be set to 0.	
		Bit 31	1 = load at run time (RAM based) 0 = load at startup time (ROM based)
		Bit 30	1 = mount volumes associated with this drive 0 = don’t mount any volume associated with this drive
		Bit 29	1 = new API entry (use 8-byte offset) 0 = old API entry (use 0-byte offset) This bit is set internally by each driver.
23–16	ATA bus ID	The bus ID where the device resides. This is the ID used to communicate with the ATA Manager.	
15–8	Device ID	The device ID within the given bus. This field is used to identify the device on a particular bus. The current and previous ATA Manager implementations assume that the device ID field is always 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 OS
- 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 0. The reference number returned must be less than 0 to be valid.
- The lower 16-bit word contains the status of the driver Open() operation. A value of 0 indicates no error.

Notify-All Driver Notification

When an error is returned from the media driver loading, the driver load function then calls the Notify-all drivers, one by one. This driver type is determined from the driver registration (–1 in the `deviceID` field of the driver registration parameter block). Unlike the media driver, this driver is notified of a device insertion by means of the callback mechanism at `accRun` time, when the manager calls the driver with an online event. Consequently, each Notify-all driver must provide a callback routine pointer in the driver registration. The driver may get a series of online event notifications during the Notify-all registration. The driver is assumed to be installed in system (for example, the INIT driver). Refer to "Notification of Device Events" on page 127 for the event opcode and the definition of the structure passed in.

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

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

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

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

ROM Driver Notification

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

Device Driver Purging

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

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

- the starting location of the driver
- the purge permissions: close(), 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)
};
```

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

Table 7-12 Purge permissions and responses

	Purge permissions		
Response	**`Close()`**	**`DrvrRemove()`**	**`DisposePtr()`**
7	√	√	√
3	√	√	
0			

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

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

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

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

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

Setting the I/O Speed

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

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

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

Error Code Summary

Table 7-13 lists two sets of error codes for ATA drivers: old error codes, used with the Macintosh PowerBook 150 and the Macintosh 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 94. If `ataPBVers` is set to 1, then the old error codes are used; if `ataPBVers` is set to 2, then the new error codes are used.

Table 7-13 ATA driver error codes

Error code (new)	Error code (old)	Error name	Error description
0	0	`noErr`	No error was 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 was not detected.
$DB44 (–9404)	$F904 (–1788)	`AT_IDNFErr`	Sector ID not-found error reported by device.
$DB45 (–9403)	$F905 (–1787)	`AT_DMarkErr`	Data mark not-found error was reported by the device.
$DB46 (–9402)	$F906 (–1786)	`AT_BadBlkErr`	A bad block was detected by the 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`	A seek error was detected by the device.
$DB4A (–9398)	$F90A (–1782)	`AT_WrFltErr`	A write fault was detected by the device.
$DB4B (–9397)	$F90B (–1781)	`AT_RecalErr`	A recalibration failure was detected by the device.
$DB4C (–9396)	$F90C (–1780)	`AT_AbortErr`	A command was aborted by the device.

continued

Table 7-13 ATA driver error codes (continued)

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

continued

Table 7-13 ATA driver error codes (continued)

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

Color Lookup Table

This appendix contains information about the color lookup table used with the flat-panel display in the Macintosh PowerBook Duo 2300c computer. Table A-1 shows the color values for each index. Index numbers are shown in hexadecimal ($0000) and decimal (0). Red (R), green (G), and blue (B) color values are shown in hexadecimal.

The first 215 entries are combinations, made up of R, G, and B values of $0000, $3333, $6666, $9999, $CCCC, and $FFFF. You should generally select colors from those 215 entries of the CLUT.

The last 40 entries are assigned to red ramp, green ramp, blue ramp, and gray ramp. The values of those last 40 entries can be dithered, either spatially or temporally, to simulate the appearance of intermediate colors. Each colored ramp consists of a single color with values of $0000, $1111, $2222, $4444, $5555, $7777, $8888, $AAAA, $BBBB, $DDDD, and $EEEE. The gray ramp, not shown, has those same values in all three color channels.

Table A-1 Color lookup table

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$0000	0	$FFFF	$FFFF	$FFFF
$0001	1	$FFFF	$FFFF	$CCCC
$0002	2	$FFFF	$FFFF	$9999
$0003	3	$FFFF	$FFFF	$6666
$0004	4	$FFFF	$FFFF	$3333
$0005	5	$FFFF	$FFFF	$0000
$0006	6	$FFFF	$CCCC	$FFFF
$0007	7	$FFFF	$CCCC	$CCCC
$0008	8	$FFFF	$CCCC	$9999
$0009	9	$FFFF	$CCCC	$6666
$000A	10	$FFFF	$CCCC	$3333
$000B	11	$FFFF	$CCCC	$0000
$000C	12	$FFFF	$9999	$FFFF
$000D	13	$FFFF	$9999	$CCCC
$000E	14	$FFFF	$9999	$9999
$000F	15	$FFFF	$9999	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$0010	16	$FFFF	$9999	$3333
$0011	17	$FFFF	$9999	$0000
$0012	18	$FFFF	$6666	$FFFF
$0013	19	$FFFF	$6666	$CCCC
$0014	20	$FFFF	$6666	$9999
$0015	21	$FFFF	$6666	$6666
$0016	22	$FFFF	$6666	$3333
$0017	23	$FFFF	$6666	$0000
$0018	24	$FFFF	$3333	$FFFF
$0019	25	$FFFF	$3333	$CCCC
$001A	26	$FFFF	$3333	$9999
$001B	27	$FFFF	$3333	$6666
$001C	28	$FFFF	$3333	$3333
$001D	29	$FFFF	$3333	$0000
$001E	30	$FFFF	$0000	$FFFF
$001F	31	$FFFF	$0000	$CCCC
$0020	32	$FFFF	$0000	$9999
$0021	33	$FFFF	$0000	$6666
$0022	34	$FFFF	$0000	$3333
$0023	35	$FFFF	$0000	$0000
$0024	36	$CCCC	$FFFF	$FFFF
$0025	37	$CCCC	$FFFF	$CCCC
$0026	38	$CCCC	$FFFF	$9999
$0027	39	$CCCC	$FFFF	$6666
$0028	40	$CCCC	$FFFF	$3333
$0029	41	$CCCC	$FFFF	$0000
$002A	42	$CCCC	$CCCC	$FFFF
$002B	43	$CCCC	$CCCC	$CCCC
$002C	44	$CCCC	$CCCC	$CCCC
$002D	45	$CCCC	$CCCC	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$002E	46	$CCCC	$CCCC	$3333
$002F	47	$CCCC	$CCCC	$0000
$0030	48	$CCCC	$9999	$FFFF
$0031	49	$CCCC	$9999	$CCCC
$0032	50	$CCCC	$9999	$9999
$0033	51	$CCCC	$9999	$6666
$0034	52	$CCCC	$9999	$3333
$0035	53	$CCCC	$9999	$0000
$0036	54	$CCCC	$6666	$FFFF
$0037	55	$CCCC	$6666	$CCCC
$0038	56	$CCCC	$6666	$9999
$0039	57	$CCCC	$6666	$6666
$003A	58	$CCCC	$6666	$3333
$003B	59	$CCCC	$6666	$0000
$003C	60	$CCCC	$3333	$FFFF
$003D	61	$CCCC	$3333	$CCCC
$003E	62	$CCCC	$3333	$9999
$003F	63	$CCCC	$3333	$6666
$0040	64	$CCCC	$3333	$3333
$0041	65	$CCCC	$3333	$0000
$0042	66	$CCCC	$0000	$FFFF
$0043	67	$CCCC	$0000	$CCCC
$0044	68	$CCCC	$0000	$9999
$0045	69	$CCCC	$0000	$6666
$0046	70	$CCCC	$0000	$3333
$0047	71	$CCCC	$0000	$0000
$0048	72	$9999	$FFFF	$FFFF
$0049	73	$9999	$FFFF	$CCCC
$004A	74	$9999	$FFFF	$9999
$004B	75	$9999	$FFFF	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$004C	76	$9999	$FFFF	$3333
$004D	77	$9999	$FFFF	$0000
$004E	78	$9999	$9999	$FFFF
$004F	79	$9999	$CCCC	$CCCC
$0050	80	$9999	$CCCC	$9999
$0051	81	$9999	$CCCC	$6666
$0052	82	$9999	$CCCC	$3333
$0053	83	$9999	$CCCC	$0000
$0054	84	$9999	$9999	$FFFF
$0055	85	$9999	$9999	$CCCC
$0056	86	$9999	$9999	$9999
$0057	87	$9999	$9999	$6666
$0058	88	$9999	$9999	$3333
$0059	89	$9999	$9999	$0000
$005A	90	$9999	$6666	$FFFF
$005B	91	$9999	$6666	$CCCC
$005C	92	$9999	$6666	$9999
$005D	93	$9999	$6666	$6666
$005E	94	$9999	$6666	$3333
$005F	95	$9999	$6666	$0000
$0060	96	$9999	$3333	$FFFF
$0061	97	$9999	$3333	$CCCC
$0062	98	$9999	$3333	$9999
$0063	99	$9999	$3333	$6666
$0064	100	$9999	$3333	$3333
$0065	101	$9999	$3333	$0000
$0066	102	$9999	$0000	$FFFF
$0067	103	$9999	$0000	$CCCC
$0068	104	$9999	$0000	$9999
$0069	105	$9999	$0000	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$006A	106	$9999	$0000	$3333
$006B	107	$9999	$0000	$0000
$006C	108	$6666	$FFFF	$FFFF
$006D	109	$6666	$FFFF	$CCCC
$006E	110	$6666	$FFFF	$9999
$006F	111	$6666	$FFFF	$6666
$0070	112	$6666	$FFFF	$3333
$0071	113	$6666	$FFFF	$0000
$0072	114	$6666	$CCCC	$FFFF
$0073	115	$6666	$CCCC	$CCCC
$0074	116	$6666	$CCCC	$9999
$0075	117	$6666	$CCCC	$6666
$0076	118	$6666	$CCCC	$3333
$0077	119	$6666	$CCCC	$0000
$0078	120	$6666	$9999	$FFFF
$0079	121	$6666	$9999	$CCCC
$007A	122	$6666	$9999	$9999
$007B	123	$6666	$9999	$6666
$007C	124	$6666	$9999	$3333
$007D	125	$6666	$9999	$0000
$007E	126	$6666	$6666	$FFFF
$007F	127	$6666	$6666	$CCCC
$0080	128	$6666	$6666	$9999
$0081	129	$6666	$6666	$6666
$0082	130	$6666	$6666	$3333
$0083	131	$6666	$6666	$0000
$0084	132	$6666	$3333	$FFFF
$0085	133	$6666	$3333	$CCCC
$0086	134	$6666	$3333	$9999
$0087	135	$6666	$3333	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$0088	136	$6666	$3333	$3333
$0089	137	$6666	$3333	$0000
$008A	138	$6666	$0000	$FFFF
$008B	139	$6666	$0000	$CCCC
$008C	140	$6666	$0000	$9999
$008D	141	$6666	$0000	$6666
$008E	142	$6666	$0000	$3333
$008F	143	$6666	$0000	$0000
$0090	144	$3333	$FFFF	$FFFF
$0091	145	$3333	$FFFF	$CCCC
$0092	146	$3333	$FFFF	$9999
$0093	147	$3333	$FFFF	$6666
$0094	148	$3333	$FFFF	$3333
$0095	149	$3333	$FFFF	$0000
$0096	150	$3333	$CCCC	$FFFF
$0097	151	$3333	$CCCC	$CCCC
$0098	152	$3333	$CCCC	$9999
$0099	153	$3333	$CCCC	$6666
$009A	154	$3333	$CCCC	$3333
$009B	155	$3333	$CCCC	$0000
$009C	156	$3333	$9999	$FFFF
$009D	157	$3333	$9999	$CCCC
$009E	158	$3333	$9999	$9999
$009F	159	$3333	$9999	$6666
$00A0	160	$3333	$9999	$3333
$00A1	161	$3333	$9999	$0000
$00A2	162	$3333	$6666	$FFFF
$00A3	163	$3333	$6666	$CCCC
$00A4	164	$3333	$6666	$9999
$00A5	165	$3333	$6666	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$00A6	166	$3333	$6666	$3333
$00A7	167	$3333	$6666	$0000
$00A8	168	$3333	$3333	$FFFF
$00A9	169	$3333	$3333	$CCCC
$00AA	170	$3333	$3333	$9999
$00AB	171	$3333	$3333	$6666
$00AC	172	$3333	$3333	$3333
$00AD	173	$3333	$3333	$0000
$00AE	174	$3333	$0000	$FFFF
$00AF	175	$3333	$0000	$CCCC
$00B0	176	$3333	$0000	$9999
$00B1	177	$3333	$0000	$6666
$00B2	178	$3333	$0000	$3333
$00B3	179	$3333	$0000	$0000
$00B4	180	$0000	$FFFF	$FFFF
$00B5	181	$0000	$FFFF	$CCCC
$00B6	182	$0000	$FFFF	$9999
$00B7	183	$0000	$FFFF	$6666
$00B8	184	$0000	$FFFF	$3333
$00B9	185	$0000	$FFFF	$0000
$00BA	186	$0000	$CCCC	$FFFF
$00BB	187	$0000	$CCCC	$CCCC
$00BC	188	$0000	$CCCC	$9999
$00BD	189	$0000	$CCCC	$6666
$00BE	190	$0000	$CCCC	$3333
$00BF	191	$0000	$CCCC	$0000
$00C0	192	$0000	$9999	$FFFF
$00C1	193	$0000	$9999	$CCCC
$00C2	194	$0000	$9999	$9999
$00C3	195	$0000	$9999	$6666

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$00C4	196	$0000	$9999	$3333
$00C5	197	$0000	$9999	$0000
$00C6	198	$0000	$6666	$FFFF
$00C7	199	$0000	$6666	$CCCC
$00C8	200	$0000	$6666	$9999
$00C9	201	$0000	$6666	$6666
$00CA	202	$0000	$6666	$3333
$00CB	203	$0000	$3333	$0000
$00CC	204	$0000	$3333	$FFFF
$00CD	205	$0000	$3333	$CCCC
$00CE	206	$0000	$3333	$9999
$00CF	207	$0000	$3333	$6666
$00D0	208	$0000	$3333	$3333
$00D1	209	$0000	$0000	$0000
$00D2	210	$0000	$0000	$FFFF
$00D3	211	$0000	$0000	$CCCC
$00D4	212	$0000	$0000	$9999
$00D5	213	$0000	$0000	$6666
$00D6	214	$0000	$0000	$3333
$00D7	215	$EEEE	$0000	$0000
$00D8	216	$DDDD	$0000	$0000
$00D9	217	$BBBB	$0000	$0000
$00DA	218	$AAAA	$0000	$0000
$00DB	219	$8888	$0000	$0000
$00DC	220	$7777	$0000	$0000
$00DD	221	$5555	$0000	$0000
$00DE	222	$4444	$0000	$0000
$00DF	223	$2222	$0000	$0000
$00E0	224	$1111	$0000	$0000
$00E1	225	$0000	$EEEE	$0000

continued

Table A-1 Color lookup table (continued)

Index (hexadecimal)	Index (decimal)	R value	G value	B value
$00E2	226	$0000	$0000	$0000
$00E3	227	$0000	$BBBB	$0000
$00E4	228	$0000	$AAAA	$0000
$00E5	229	$0000	$8888	$0000
$00E6	230	$0000	$7777	$0000
$00E7	231	$0000	$5555	$0000
$00E8	232	$0000	$4444	$0000
$00E9	233	$0000	$2222	$0000
$00EA	234	$0000	$1111	$0000
$00EB	235	$0000	$0000	$EEEE
$00EC	236	$0000	$0000	$DDDD
$00ED	237	$0000	$0000	$BBBB
$00EE	238	$0000	$0000	$AAAA
$00EF	239	$0000	$0000	$8888
$00F0	240	$0000	$0000	$7777
$00F1	241	$0000	$0000	$5555
$00F2	242	$0000	$0000	$4444
$00F3	243	$0000	$0000	$2222
$00F4	244	$0000	$0000	$1111
$00F5	245	$EEEE	$EEEE	$EEEE
$00F6	246	$DDDD	$DDDD	$DDDD
$00F7	247	$BBBB	$BBBB	$BBBB
$00F8	248	$AAAA	$AAAA	$AAAA
$00F9	249	$8888	$8888	$8888
$00FA	250	$7777	$7777	$7777
$00FB	251	$5555	$5555	$5555
$00FC	252	$4444	$4444	$4444
$00FD	253	$2222	$2222	$2222
$00FE	254	$1111	$1111	$1111
$00FF	255	$0000	$0000	$0000

Glossary

680x0 code Instructions that can run on a PowerPC processor 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 low-speed 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.**

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.**

GeoPort A software and hardware solution for digital telecom and wide-area connectivity using the serial port.

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.

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

LocalTalk The cable terminations and other hardware that Apple supplies for local area networking from Macintosh serial ports.

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

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

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 for plug-in expansion cards in Macintosh computers.

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.

Open Transport A networking architecture that allows communications applications to run independently of the underlying network; formerly known as *Transport-Independent Interface (TII).*

PBX In the Macintosh PowerBook Duo 2300c computer, the custom IC that provides the interface between the PowerPC 603 bus and the I/O bus.

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 Curio IC that provides an interface to the serial data ports.

Small Computer System Interface (SCSI) An industry standard parallel bus protocol for connecting computers to peripheral devices such as hard disk drives.

Versatile Interface Adapter (VIA) The hardware interface for system interrupts that is standard on most Macintosh 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

This Apple document was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof and final pages were created on an Apple LaserWriter Pro 630 printer. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Allen Watson III

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATORS
Deborah Dennis, Shawn Morningstar, and Sandee Karr

Special thanks to Rodney Amen, Lorenzo Dunn, Tom Llewellyn, and Nancy Schmitt