INSIDE MACINTOSH

Modular I/O

WWDC Release

May 1996

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

About the I/O Architecture

Contents

This chapter provides an overview of the I/O architecture of Mac OS 8. The I/O architecture is designed to improve the user experience by providing superior performance, better responsiveness, and increasingly robust systems, and by supporting the advancements inherent in a microkernel-based operating system. It improves the developer experience by increasing the predictability of I/O responsiveness, by simplifying driver development, by minimizing and localizing hardware dependencies in software, and by providing an improved Device Manager.

You need to understand the framework that the I/O architecture provides for innovation and how it affects compatibility with both hardware and software products if you are one of the following types of developers:

- If you are a Mac OS licensee, you need to understand the I/O architecture to be certain that devices you incorporate into your hardware product will operate with Mac OS 8 and to understand how software can be loaded into your product when it is turned on.
- If you are a hardware vendor who makes NuBus™ or PCI cards, Apple Desktop Bus (ADB) devices, GeoPort pods, or other hardware devices that operate with Mac-compatible computers, you need to know how to create software that allows access to your product.
- If you produce software products such as network protocol implementations, file system implementations, or virtual device drivers to extend the capabilities of Mac OS 8, or if you develop utilities such as driver installers, hard disk formatting and partitioning packages, or disk recovery and repair products, you need to understand the I/O architecture to determine if you need to modify your software product to run on Mac OS 8.
- If you are an application developer whose application writes to or otherwise manipulates devices, you need to understand how to take advantage of the new features in the I/O architecture and how to enhance your application's compatibility with future versions of Mac OS.

This chapter briefly introduces the I/O architecture of Mac OS 8. Then it discusses

- selected aspects of I/O families and their plug-ins
- short-term and long-term design goals of the I/O architecture
- I/O support services in Mac OS 8
- family activation models

- the name registry and its role in the I/O subsystem
- compatibility issues for device driver writers and application developers

Before reading this chapter, you should be familiar with the architecture of Mac OS 8, including tasks and processes, memory organization and protection, synchronization and notification methods, and microkernel messages. You can find information about these topics in *Microkernel and Core System Services*.

Introduction

With Mac OS 8, the workings of the lowest levels of the Mac OS change from what they were in previous versions of system software. The implementation of a microkernel-based, preemptive, multitasking operating system has significant implications for developers creating drivers and other I/O services for the Mac OS and for applications that use them.

- Software running in user mode and software running in supervisor mode have no direct access to each other's data. Because drivers run in supervisor mode and applications normally run in user mode, drivers are protected from applications and vice versa. An application gets access to driver services only through an I/O family's client programming interface.
- I/O devices are not directly accessible to applications, nor are they vulnerable to application error. Applications get access to hardware only through an I/O family's client programming interface.
- The context within which a driver runs and the method by which it interacts with the system are defined by the I/O family to which it belongs.

The notions of I/O family, client, and I/O plug-in are fundamental to the I/O architecture of Mac OS 8. An **I/O family** is a collection of software components that provide a distinct set of services to the system. For example, the SCSI family and its SCSI interface modules (SIMs) provide access to devices connected to SCSI buses; the file systems family and its volume-format plug-ins provide support for different file systems; the Open Transport family and its Data Link Provider Interface (DLPI) device drivers provide network services. Often, a family is associated with a set of devices that have similar characteristics, such as display devices or ADB devices.

A **client** of an I/O family is any software that requests services offered by the family. A family's clients can include applications, other I/O families and their

plug-ins, server programs, and system software. A given family's clients and its plug-ins are mutually exclusive sets of software components. Each family provides two programming interfaces: one for its clients and one for its plug-ins.

An **I/O plug-in** is a dynamically loaded piece of software that provides particular implementation of the service offered by a family. For example, within the file systems family, a plug-in implements services particular to a volume format such as HFS or DOS FAT. You can extend Mac OS 8 by writing new I/O plug-ins.

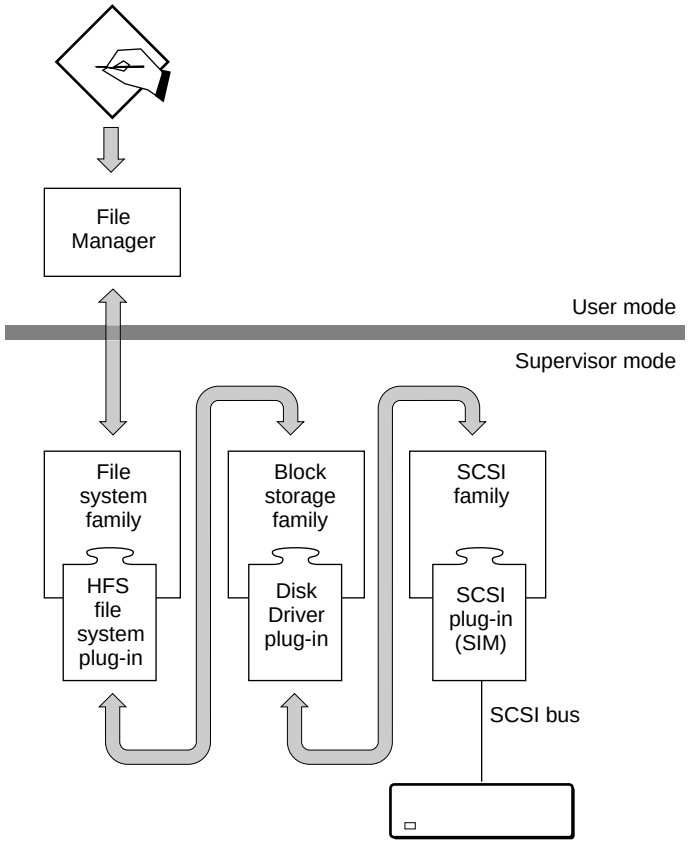
I/O plug-ins are a superset of device drivers. A device driver is a software component that communicates with and controls a hardware device. While all device drivers are I/O plug-ins, not all I/O plug-ins are drivers. For example, the partitioning plug-in defined by the block storage family, for example, is not a device driver.

Note
In documents other than those that describe the I/O subsystem or some part of it, the term *plug-in* may refer to dynamically loaded software that is not related to I/O. For convenience, the rest of this chapter frequently uses the simple term *plug-in* rather than *I/O plug-in* to refer to an I/O plug-in. ◆

In Mac OS 8, code that executes in supervisor mode, as families and plug-ins do, is *trusted*. A failure in one of these software subsystems can cause complete system failure. However, failure of any particular application does not affect the ability of the I/O subsystem and other microkernel-level services to continue serving other clients. The I/O subsystem is insulated from application error.

Figure 1-1 illustrates an example of the relationship between an application, several I/O families, and their plug-ins. An application requests services of an I/O family through the client programming interface, such as the File Manager. Typically, the application request is converted to a microkernel message, shown in the figure as gray arrows, to cross the boundary between the user mode and supervisor mode environments and be delivered to the family.

Figure 1-1 High-level view of an application, I/O families, and plug-ins



Note that Figure 1-1 shows three I/O families that work together to complete a service request. The application makes the service request, which then moves through the file systems family, the block storage family, and the SCSI family. This particular routing of the request does not imply any hierarchical relationship among families—all families are peers of each other.

In introducing the concepts of family and plug-in, the I/O architecture of Mac OS 8 formalizes existing programming practices. For example, when an application accesses the services of a video device through the Display Manager, it is calling the display family. The Display Manager is tailored to the

needs of video devices. Likewise, when an application calls the Sound Manager, it is calling the sound family. The family concept in the I/O architecture explicitly acknowledges that similar devices share many characteristics and needs. Therefore, it provides programming interfaces tailored to the needs of specific device families. These specially tuned sets of services allow drivers for a given family to be as simple as possible.

Apple will provide a number of I/O families in its first release of Mac OS 8, including, but not necessarily limited to, the following:

ADB family	NVRAM family
ATA family	Open Transport family
block storage family	PC card family
Device Manager family	PCI family
display family	pointing family
file systems family	real-time clock family
keyboard family	SCSI family
NuBus family	sound family

With the first release of Mac OS 8, Apple will provide plug-ins for the families listed above. You are encouraged to develop new plug-ins.

I/O Families

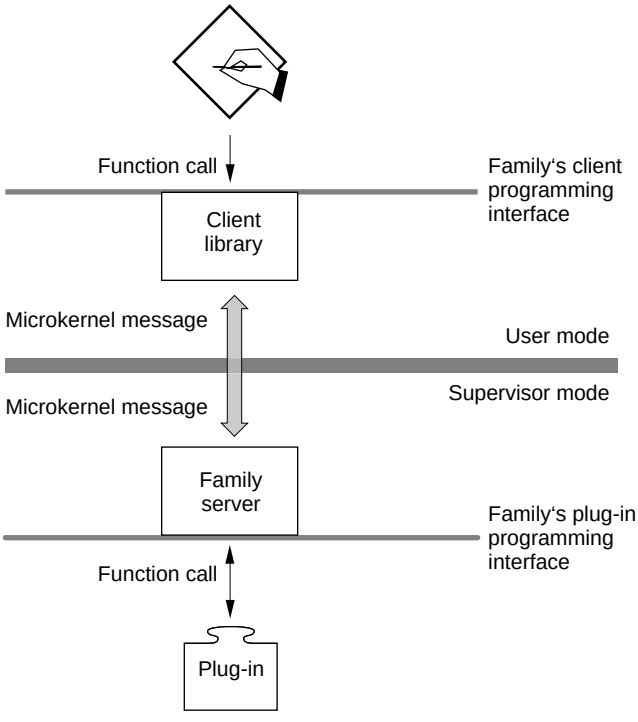
An I/O family typically consists of the following software components:

- a programming interface for the family's clients
- a client library that implements the client programming interface and sends client requests to the family server
- a programming interface for the family's plug-ins
- a family services library that implements the plug-in programming interface and that, optionally, provides other services to family plug-ins
- a family server that receives client requests and, usually, calls a plug-in to process them

- a family expert that maintains awareness of family-specific devices or services available on a given computer
- the set of family plug-ins

Figure 1-2 provides a simplified view of how a family client code library, a family server, and a plug-in are related to each other; of how user mode software and supervisor mode software are separated; and of the distinct programming interfaces that a family provides for its clients and for its plug-ins.

Figure 1-2 I/O family software diagram



Family Client Programming Interface

Each family provides a programming interface for its clients. The interface provides clients with access to services specific to that particular family.

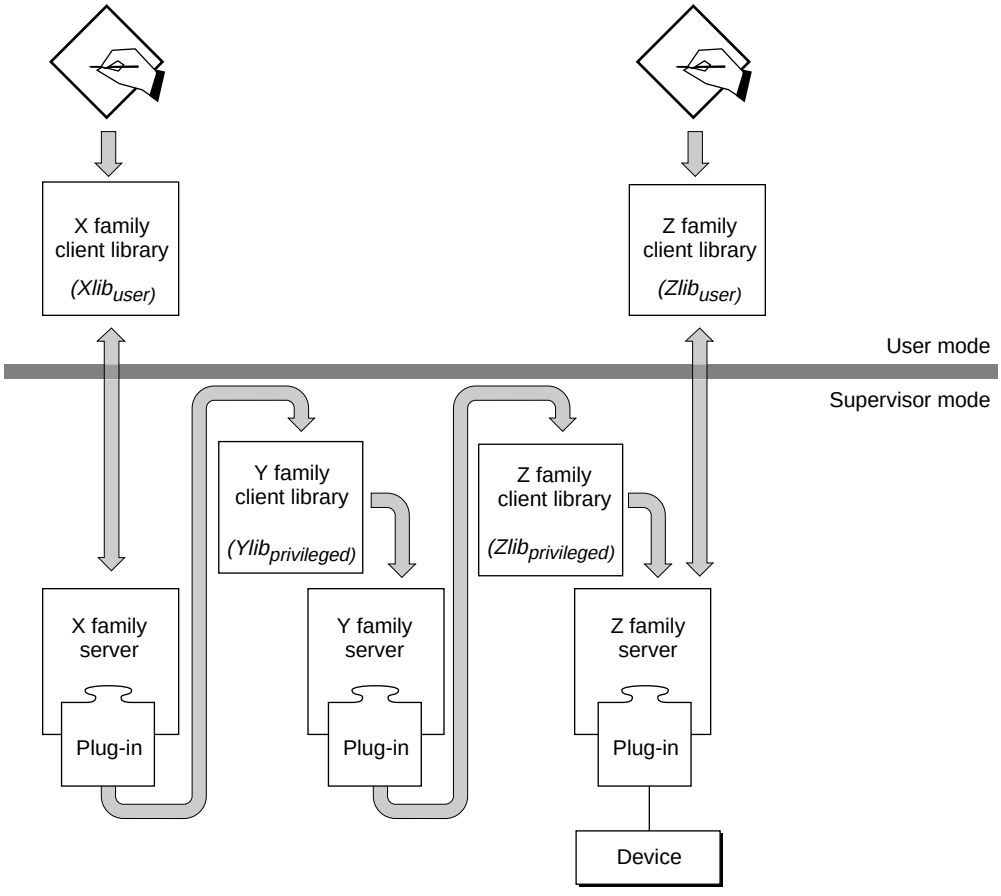
Note
The term *family programming interface* has been used in the past to refer to the client programming interface and to distinguish it from 1) other interfaces available in Mac OS 8, such as those provided by the microkernel or the Toolbox, and 2) the family's plug-in programming interface. ◆

A family's client library implements the client programming interface, and forwards client requests for service to the family server. Typically, it translates client function calls into microkernel messages and sends the messages to the family's server for processing. The structure of the messages and the protocols governing their use is defined by the family.

User-Mode and Supervisor-Mode Client Libraries

To make certain optimizations possible, a family may provide two versions of its client library, one for user-mode clients and one for supervisor-mode clients, as illustrated in Figure 1-3.

Figure 1-3 User-mode and supervisor-mode client libraries



The boxes labeled $Xlib_{user}$ and $Zlib_{user}$ represent the libraries that support the client programming interfaces of the X and Z families for user-mode clients. The boxes labeled $Ylib_{privileged}$ and $Zlib_{privileged}$ represent the Y and Z family libraries available to supervisor-mode clients. The Z family provides a user-mode version and a supervisor-mode version of its client library.

Both the user-mode and the supervisor-mode versions of the libraries present the same interface to clients. Mac OS 8 distinguishes between the user-mode and supervisor-mode versions to permit optimization of the supervisor-mode libraries in some instances. For example, operations that must be implemented in the user-mode library, such as copying data across address space boundaries, may be unnecessary in the supervisor-mode library. In some instances, the user-mode and supervisor-mode versions may be the same.

Connection-Based Services

Programming interfaces for I/O family clients are connection-based. All clients need a connection ID to get access to services or devices through the family. Typically, a family provides a function to open a connection to a service or device specified by a client. The function returns a connection ID. Thereafter, the client passes the connection ID when calling into the family. Families also provide a function to close a connection. When a client no longer needs a service or device, it closes the connection to release any resources held by the family to support the connection.

Family Plug-in Programming Interface

A family's programming interface for its plug-ins defines two sets of functions and data structures:

- the functions and data structures a plug-in exports to the family
- the functions and data structures a family exports to its plug-ins

A **family services library** implements the functions and data structures that the family exports for its plug-ins, and defines the methods by which the family and its plug-ins exchange data. Functions exported by a family can be characterized as one of two general types:

1. Functions that a plug-in must call when certain events, such as an I/O completion, occur. Functions of this type implement communication between the family and its plug-in. For example, they may use a notification mechanism to unblock a family task and make it eligible for execution, or may maintain state information needed by a family to dispatch and manage requests. All family services libraries contain functions of this type.
2. Functions that provide family-specific services to plug-ins and that require no task context switch. These functions execute in the context of the plug-in. Often, they help the plug-in manipulate data. For example, the family

services library for the display family may provide functions that deal with vertical blanking. A family services library that provides commonly needed functions simplifies the development of that family's plug-ins. Not all family services libraries contain this type of function.

Family Servers

A **family server**, which always runs in supervisor mode, receives, processes, and responds to service requests from an I/O family's clients.

How a server responds to a request depends on the family's activation model. The server is, in essence, the implementation of the model. A server, for instance, may put an incoming request in a queue or call a plug-in directly to service a request. (For an overview of three basic activation models, see "Activation Models" (page 1-26).)

Typically, the client library and the family server use microkernel messages to communicate, and as a result, the server supports a message port. The server uses an accept function or a task to wait for messages on the message port. The accept function or task dispatches incoming requests by calling entrypoints in the family code.

I/O Path Communications

Although microkernel messages are the usual method of communicating I/O requests between the client libraries and the server for a given family, between different families, and between plug-in *x* and family *z*, other communication mechanisms are possible. The choice is up to the family designer.

The use of microkernel messages facilitates the development of I/O families by providing a very easy programming model. For family designers, it offers a straightforward interfamily communication mechanism, very fast and efficient. In addition, the use of microkernel messages permits greater independence of family activation models.

When I/O families use microkernel messages, performance improvements in the messaging service are automatically reflected in the I/O subsystem.

Whatever the communication method, it is completely opaque to a client requesting a family service. Clients simply use the client programming interface to make requests of the family and its plug-ins.

Administrative Message Ports

Each I/O family provides an administrative message port (AMP) through which the family receives all administrative information. An AMP is a standard microkernel message port. Microkernel messages received at the AMP inform families of such things as changes in hardware configuration (if the family supports hot swappable devices), power management requirements, and access control.

A family designer can choose to receive normal I/O requests and administrative messages at the same port. In that case, the family must service administrative messages before I/O requests. (A family using a single message port defines separate message objects for I/O requests and for administrative messages and uses message type masks to differentiate between I/O requests and administrative messages.)

A family must provide a task to process messages received on the AMP. Families that use an accept function to receive I/O requests need to create a task to handle administrative messages.

High-Level and Low-Level Families

I/O families are characterized as being high level or low level, or both. The characterization refers to the role the family expert plays at boot time when the system hardware is being recognized and, after the system is up and running, when a device is added or removed.

A **family expert** is the code within a family that maintains knowledge of the set of family-controllable devices and plug-ins for a given Mac-compatible system. To do this, the expert uses the Driver and Family Matching (DFM) service, Device Notification Service (DNS), and the name registry. For a brief look at these topics, see “Driver and Family Matching” (page 1-22), “Device Notification Service” (page 1-22), and “Name Registry” (page 1-34).

Although this section discusses the expert’s roles in terms of adding a device to a running system, the basic ideas apply to the removal of a device and to the boot process as well.

A **low-level family** can detect when a device that can be controlled by the family is added or removed. When a low-level family expert detects a new device, it creates an entry in the device tree portion of the name registry and uses DNS to send a notification of the event to DFM.

DFM locates plug-ins that match the new device and adds pointers to the plug-ins to the name registry entry representing the new device. Then DFM sends a notification to all software that previously registered interest in receiving such notifications.

A **high-level family** registers to receive new-device notifications of devices that the family can control. After receiving such a notification, a high-level family expert inspects the new name registry entry for the device and examines the plug-ins located by DFM for the new device. Then it selects the plug-in it deems most suitable to manage the device.

The cooperation between the experts of low-level and high-level families and DFM enables Mac OS 8 to respond gracefully to changes in system configuration. As a result, the set of plug-ins known to and available through a family remains current with the actual hardware available to a system.

To summarize, the expert of a low-level family knows how devices are connected to the system, it scans hardware, and it adds and alters entries in the name registry. The expert for a high-level family does none of these things—it is insulated from knowledge of physical connectivity. High-level family experts examine specific entries in the name registry after being notified of events of interest and select the best plug-in to manage a new device.

A family can be simultaneously high level and low level. Consider this example: The SCSI family is a low-level family; the SCSI expert can detect when a device is added to or removed from the SCSI buses it knows about. It sends notifications when it discovers a new device. Suppose the SCSI expert registers to receive notifications of new SCSI buses. Further suppose that on a computer with a PC card bay, a user inserts a PC card with a SCSI controller on it. The PC card family is the low-level family in this instance. It detects the newly inserted card and sends a new-device notification. The SCSI family, by virtue of registering for and receiving the new-device notification, is also a high-level family.

I/O Plug-ins

An I/O plug-in is a dynamically loaded piece of software that provides a particular implementation of the service provided by an I/O family. For example, within the file systems family, a plug-in implements file-system-specific services. The plug-ins understand a particular volume

format such as HFS or DOS FAT. But file systems family plug-ins don't understand how to get data from a physical device. To read data, a file systems family plug-in calls the block storage family. The block storage family then calls one of its plug-ins—a media-specific driver such as a CD-ROM driver or a hard disk driver—that in turn calls another family, such as the SCSI family, to access data on a given physical device. (Figure 1-1 (page 1-6) illustrates these relationships.)

The following statements apply to all I/O plug-ins:

- They have a layered structure. Most of their work is done at task level. A small amount of work may be done by interrupt handlers. The layered structure allows plug-in code to be partitioned so that it works well within the Mac OS 8 architecture.
- They can be written in a high-level language.
- They must be compiled into native PowerPC™ code.
- They are packaged as Code Fragment Manager fragments.
- They cannot call Toolbox routines or other non-reentrant services.
- They must conform to their family's activation model.
- They run in supervisor mode and therefore have access to the microkernel's protected memory space.

A family calls a plug-in to service a request made by a family client. A plug-in usually has two parts that do the following:

- The main code section runs at task level. It is here that the plug-in does most of its work. All plug-ins must have a main code section.
- A hardware interrupt handler services hardware interrupts if the plug-in responds to a physical device. When handling an interrupt, a hardware interrupt handler should perform only essential functions and defer all other work to the plug-in's main code section or a secondary interrupt handler. The plug-in programming interface specifies how interrupts are managed within a family. Not all plug-ins need a hardware interrupt handler.

A plug-in should make no assumptions about particular hardware settings or configurations. However, it should never attempt to obtain device configuration information directly through programming interfaces, such as the Resource Manager or the File Manager. A plug-in can obtain configuration information in several ways.

- To get static configuration information, it can read information stored in the name registry.
- When a family client calls the family to report a dynamic configuration change, the family forwards the information to a plug-in through the plug-in programming interface.
- A plug-in can use another family's client programming interface to obtain some types of configuration information. For instance, a video plug-in may call NVRAM family client functions to obtain video mode information stored in NVRAM prior to the last system reboot.

Family plug-ins must conform to the activation model defined by the family and provide the code and data exports described by family documentation. Other chapters in this book describe the required interfaces for specific I/O families.

Although the activation model and the required code and data exports for each family are family specific, the packaging for all plug-ins is the same—they are all Code Fragment Manager fragments.

Note

Complete packaging requirements for plug-ins are not defined in this Developer Release. For example, currently each fragment must reside in a separate file. At a later date, you may be able to store multiple fragments in a single file. ◆

The standard family and plug-in definitions cover most cases of I/O component development. However, there are exceptions to the model. The next sections describe two.

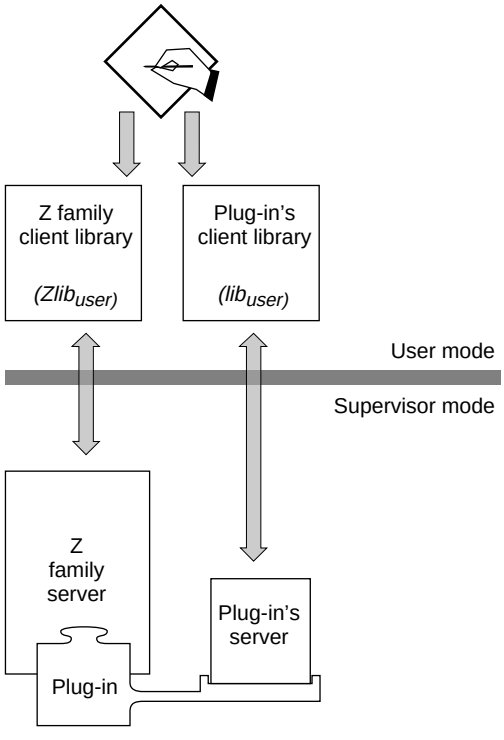
Extending Client Programming Interfaces

A plug-in may provide an additional interface specific to itself so that it can provide services beyond those available through its family's client programming interface. This capability is useful in a number of situations. Take, for example, a block storage plug-in that controls a CD-ROM device. Many CD-ROM devices provide an interface that allows knowledgeable application software to control audio volume and to play, pause, stop, and so forth. The block storage family does not include functions to control audio volume and playback. Such added capabilities can be provided through a plug-in-specific programming interface.

Most family interfaces provide some level of extensibility to the family's plug-ins. For example, the Device Manager allows extensible sets of control and status selectors that may be used to gain device-specific information and control. And Open Transport device drivers may receive special calls to extend the device information and control. This kind of device extension within the family framework is not changed with the I/O architecture of Mac OS 8. If, however, a device wishes to export extended services outside the family framework, it needs to provide a separate message port and an interface library for that portion of the device driver.

Figure 1-4 illustrates a plug-in that provides an extended programming interface—it offers features in addition to those available to clients through the family's client programming interface.

Figure 1-4 Extending a client programming interface



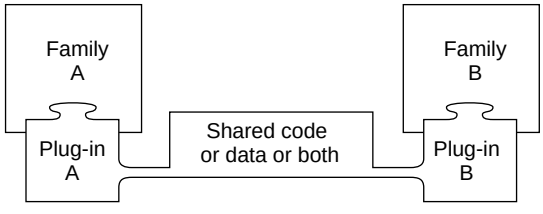
To make its extra services available, the plug-in must provide the additional software shown in Figure 1-4:

- the client library that offers the extended features (lib_{user})
- the server with its message port and the code that implements the extra features

Sharing Code and Data Between Plug-ins

Two or more plug-ins can share data or code, or both, regardless of whether the plug-ins belong to the same family or to different families. Sharing code or data is desirable when a single device driver wishes to subscribe to two or more families. Such a driver needs a plug-in for each family. These plug-ins can share libraries that contain information about the device state and common code. Figure 1-5 illustrates two plug-ins that belong to separate families and that share code and data.

Figure 1-5 Plug-ins that share code and data



Plug-ins can share code and data through Code Fragment Manager fragments (shared libraries). The Code Fragment Manager allows you to instantiate independently plug-ins that share code or data without encountering problems related to simultaneous instantiation. The first plug-in to be opened and initialized gets sole access to the shared libraries. When the second plug-in is opened and initialized, it establishes a new connection to the shared libraries. From that time forward, the two plug-ins contend with each other for access to the shared libraries.

Sharing code or data is desirable in certain special cases. For example, some of the special-case solutions provided on System 7 use two or more separate device drivers that use shared data as a communication mechanism. Typically,

special-case solutions install a set of devices and a set of special drivers. The closely coupled devices use a high-speed data path to move data between them. For example, a video input device puts video data in a shared buffer; subsequently, a video compression device reads and compresses the data it finds in the shared buffer. Access to the high-speed data path via the shared buffer is synchronized by solution-specific mechanisms. In Mac OS 8, this type of solution can be implemented as a vendor-supplied family and its plug-ins.

Design Goals of the I/O Architecture

The next two sections describe the short-term and long-term design goals of the I/O architecture of Mac OS 8.

Short-Term Design Goals

In the first release of Mac OS 8, the I/O architecture is targeted to meet the following design goals:

- **End-user flexibility.** Mac OS provides end users with tremendous value that is directly attributed to the flexibility and adaptability of its I/O subsystem. For example, its plug-and-play capability and dynamic monitor configuration are features that are simply not possible with many I/O architectures. The I/O architecture is designed to provide these end-user features and to retain this flexibility in Mac OS 8.
- **Performance.** The I/O architecture favors lower-latency responses over higher bandwidths to provide greater responsiveness to users. To help achieve this goal, all drivers and all their support services are native. Additionally, very little code is permitted to run at the hardware-interrupt level. Although the architecture does not guarantee the best performance for burst and single-stream high-bandwidth clients, the Mac OS 8 implementation will produce much better throughput results than are available in System 7. The I/O architecture provides support for the real-time needs of MIDI, sound, GeoPort, and QuickTime and enables implementations that meet or exceed the performance of competing platforms.
- **PCI driver compatibility.** The I/O architecture accommodates the I/O system of PCI-based Mac-compatible computers. Drivers compliant with the

specification for driver development contained in the document *Designing PCI Cards and Drivers for Power Macintosh Computers* will continue to function well within the I/O model of Mac OS 8. In addition, Mac OS 8 seeks to provide binary compatibility with PCI ROM-based video and network drivers developed in accordance with the specification for native drivers described in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

- **Reliability, availability, and serviceability.** In Mac OS 8, the I/O subsystem works as expected and continues to work acceptably in the face of failures in particular areas. For instance, disk I/O continues to work if a failure in the serial hardware occurs. When failures do occur, the architecture provides support for analysis and corrective measures by the user and by support organizations.
- **Resource allocation and control.** Having limited resources, I/O components distribute those resources fairly among themselves. In particular, the first driver loaded cannot consume resources such as memory, message ports, timers, interrupt latency, or bus bandwidth in a way that prevents subsequent drivers from loading or operating correctly. Configurations that cannot work because their needs are mutually exclusive are recognized and reported in a meaningful way.
- **Power management.** Required for battery-powered systems such as PowerBook computers, the need for integrated power management is increasing for all systems. The I/O architecture provides an infrastructure to enable optimal power management in diverse systems.
- **Extensibility.** The I/O architecture enhances the ability of OEMs to create Mac-compatible hardware and peripherals. It is intended that all hardware-dependent software fall into one of two categories:
  - software based on clearly defined hardware invariants, such as big-endian addressing and the PowerPC 601, 603, and 604 processors
  - software that is dynamically loadable at system startup time, such as I/O families and their plug-ins

Long-Term Design Goals

In subsequent releases of the Mac OS, the I/O architecture is targeted to meet these additional design goals:

- **Multiprocessor support.** High-quality support for a limited number of tightly coupled, cache-coherent processors is a long-term goal of the

architecture. While revisions to the architecture may be desirable for multiprocessor systems, conforming I/O components should be compatible within multiprocessor versions of the architecture.

- **Real-time I/O support.** The architecture specifies basic support for real-time I/O needs, largely as a subset of the resource allocation and control mechanisms provided by the architecture. Families and plug-ins are prioritized according to their needs to better support real-time clients.
- **Improved reliability, availability, and serviceability (RAS).** RAS is the natural successor to the Mac OS plug-and-play capability. The addition of RAS to Mac OS provides users, system administrators, and technicians with a broad set of tools for maintaining a Mac OS system, resulting in lower training and support costs. RAS is one of the mechanisms by which Mac OS will maintain its lead as the easiest and most configurable system available.
- **Visual system administration.** Enabling end users, system administrators, and support staff to examine and manipulate the configuration of a specific system is a natural extension to the benefits of RAS support.
- **Scalable to future technologies.** Mac OS 8 provides sufficient architectural integrity to ensure that implementations of technologies that are not quite available today are obtainable on desktop platforms. ATM and infrared networking and Firewire bus connectivity are examples of such technologies.
- **Distributed computing.** As system performance increases, it is increasingly reasonable to provide access to devices that are not attached directly to the CPU on which an application is running. For example, with high-cost, high-speed networks, it is possible today for an application running on a given computer to capture video via a frame-grabbing card plugged into another networked computer. As networking costs decrease, distributed services become feasible on increasing numbers of desktop systems. Distribution of I/O subsystems across a suitable network is a long-term goal of this architecture.
- **Universal booting.** A single system image that boots on all hardware configurations that support Mac OS 8 is a goal of the architecture. In addition, these systems will support both minimal and third-party customized installations of Mac OS.

I/O Support Services

This section briefly describes the I/O support services available in Mac OS 8. These services are available to all families and plug-ins—that is, they are not specific services for different classes of devices, such as serial devices or video display monitors.

Driver and Family Matching

The Driver and Family Matching service (DFM) enables Mac OS 8 to respond to changes in hardware configuration without disturbing the microkernel's operation. DFM finds and loads hardware-specific software components, such as plug-ins and family experts, at system boot time. I/O families and their plug-ins must export certain data structures that DFM needs to perform its functions. At boot time, DFM locates the right software components for the hardware available on a given computer (regardless of whether the hardware-specific software is stored in ROM, on disk, or both), loads the code into memory, adds information about the software to the proper entry in the name registry, and generates notifications to interested parties.

If the hardware configuration changes dynamically after Mac OS 8 is up and running, DFM interacts with the Device Notification Service and family experts to locate and make available all software needed to manage the hardware.

See "Driver and Family Matching" (page 2-3) for more information on DFM.

Device Notification Service

The Device Notification Service (DNS) provides support for hot swappable devices such as PC cards. The I/O architecture can support—through family experts, DFM, and DNS—dynamic changes in connectivity to devices that may appear and disappear at any time. This feature allows a user to insert and remove devices such as disk driver card or modem card without powering down and restarting the computer.

DNS defines a set of event notifications and a programming interface for its users, referred to as producers and consumers. low-level family experts that notify DNS about an event are producers. High-level family experts that

register to receive certain types of notifications are consumers. A producer need not know about the consumers that receive the notifications it produces. DNS is responsible for sending notifications to registered consumers. A consumer receives notifications in microkernel messages sent to its administrative message port.

Conglomerate Libraries for Plug-ins

In addition to the family-specific services available from a family services library, a plug-in may need generic system services such as interrupt registration, timing facilities, synchronization services, and secondary interrupt-handling capabilities. Mac OS 8 provides these base-level services in a number of shared libraries.

Rather than require a plug-in developer to specify many libraries at link time, each family provides a **conglomerate library** that contains the family services library and the generic system services needed by its plug-ins. Thus, at link time, you simply specify the conglomerate library provided by the family to which your plug-in belongs.

Following is a list of some of the libraries, in addition to the family services library, that may be included in a family's conglomerate library:

DeviceManagerSupport
DriverSupport
DriverSynchronization
Interrupts
Kernel
Synchronization
Timing

Family-provided conglomerate libraries replace the Driver Services Library provided for the first PCI drivers. Drivers that link to the Driver Services Library will continue to work in Mac OS 8, provided that they adhere to the specifications in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

Booting Services

The I/O architecture provides a method for loading and launching the system software. The microkernel booting architecture maintains the Mac OS user experience at system startup. The user is not required to build a system tailored for the hardware that the system will run on. Many users may choose to install hardware support for a large class of devices that might be connected to their

computers. For those users, the system finds the right support software at startup time and configures that software into a runnable system without user intervention. For more information on booting, see "Booting Services" (page 9-3).

Power Management

The I/O architecture provides mechanisms for power state transitions within the system, such as bringing the system up, shutting it down completely, and maintaining a sleep state. The power management service in Mac OS 8 coordinates the power needs of individual devices in the system as well the system as a whole. The same power management service is available on all computers that run Mac OS 8, both portable and desktop machines.

The power management service supports five system power modes:

- **Power management disabled mode.** All devices in the system are turned on, and the system never automatically enters any power-saving mode.
- **Normal mode.** The core system is available for service requests. The power management service can control the power consumption of individual devices based on the use of those devices and, in some cases, turn off devices that are idle for user-selectable periods of time. If the entire system is idle for a user-selectable period of time, it transitions into a user-selectable low-power mode.
- **Sleep mode.** The contents of memory are preserved, but active processing is halted. Returning to normal mode is very quick.
- **Hibernate mode.** The system state is written to disk, and the computer is turned off. When the computer is turned on, the saved state is read into memory, and users can continue where they left off—applications that were open before going to hibernate mode are again running and documents that were open are again open. This process takes longer than returning to normal mode from sleep mode, but is faster than a full reboot.
- **Power-off mode.** The entire system is powered down, and no state information is saved.

The power management service decides when an individual device or the system as a whole should change its power state. I/O families and plug-ins never initiate a power state change.

The family's programming interface for its plug-ins provides mechanisms by which the family asks its plug-in to provide information about the power state capabilities of the plug-in and the device it manages, such as

- the power states that a given device can support
- the relative power consumption of each state
- the service that the device can provide in each state

The plug-in programming interface also provides mechanisms by which the family sends power-state change requests and other power management events to the plug-in.

The power management service interacts with I/O families to manage the power consumption of devices in the system. The family provides the power state capability information it retrieves from its plug-ins to the power management service.

Families that monitor individual device use—such as the block storage, Open Transport, keyboard, and pointing families—periodically report the information to the power management service. Plug-ins provide power management data as required by their families, rather than actively monitoring their devices. The family, by directing the plug-in, controls the monitoring.

The power management service maintains information on the power state and power requirements of each device whose power it can manage. When the service determines that a device or the system as a whole is idle, it directs the family (or families) to reduce power consumption. The family, in turn, directs its plug-in to change the power state of the device.

The power management service seeks to manage the power consumption of the system at a fine granularity to prolong battery life in portables and to provide better power savings for desktop machines. Plug-in designers should keep this in mind and take a device's power state capabilities into account when designing their software.

A plug-in should be designed to handle power to its device being turned off and on and to allow access to the device to continue as if power had never been turned off. Even though the device may not include the capability of turning itself off and on, the system as a whole can be turned off and on (in transition to or from hibernate mode).

Activation Models

A family's **activation model** consists of the tasking and communication implementation choices made by the family designer. An activation model defines both the implementation of the family software and the environment within which a family's plug-ins execute. It defines the relationship between family code and its plug-ins, including such things as

- the tasking model that a family uses
- the opportunities for execution that the family provides to its plug-ins and the context of those opportunities (for instance, a plug-in might be called at task level or at secondary interrupt level, or both)
- the knowledge about states and processes that a family and its plug-ins are expected to have
- the portion of the service requested by the client that is performed by the family and the portion that is performed by the plug-ins
- the required characteristics of plug-ins, such as whether the plug-in blocks or returns an error when it encounters resource exhaustion
- the methods by which data is communicated between the family server and a plug-in, memory is allocated, interrupts are registered and serviced, and timing services are provided

If you want to develop a new I/O family, you need to design an activation model that best suits the needs of your I/O family and then implement the family server in light of that activation model. If you want to develop a new plug-in, you need to understand the activation model used by the family to which your plug-in belongs.

This section describes three basic family activation models used by Mac OS 8 I/O families. Each model provides a distinctly different environment for the plug-ins to the family, and different implementation options for the family software. The activation models discussed here are

- the single-task model
- the task-per-plug-in model
- the task-per-request model

Many variations of and hybrid approaches to the activation models discussed here are possible. The choice of activation model is left to the family designer. The selected models are simply examples of how you can implement a family.

To provide the asynchronous or synchronous behavior desired by the family client, the three activation models discussed here use microkernel messages as the interface between the client libraries and the family servers. The activation models require the family to provide a task context for asynchronous I/O requests from clients.

The family designer's choice of activation model limits plug-in implementation options. For example, the activation model defines the interaction between a driver's hardware interrupt handler and the family runtime environment in which the main driver code runs. A plug-in must conform to the activation model employed by its family. As a result of this well-defined environment, plug-in development is simplified.

You will find it hard to understand the discussion of activation models without some understanding of microkernel messages, tasks, and interrupt mechanisms in Mac OS 8. For information about these topics, see *Microkernel and Core System Services*.

Single-Task Model

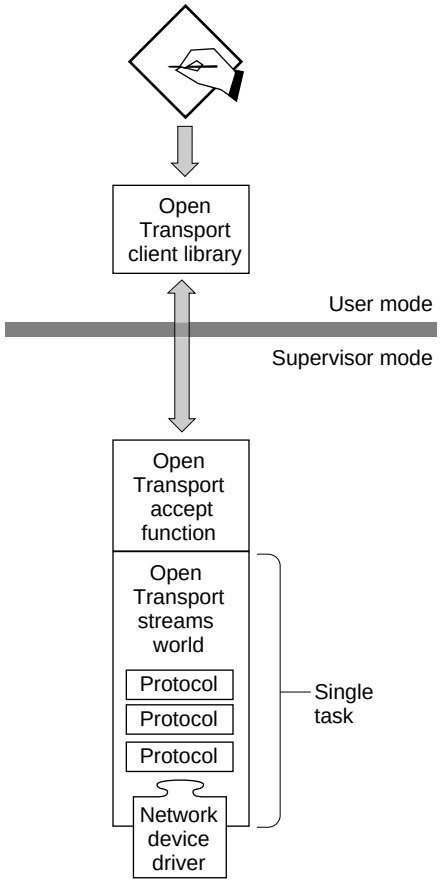
In the single-task activation model, a single monolithic task is fed from above by a request queue and from below by interrupts delivered by the plug-ins. The family's client library sends client requests to an accept function that queues the request for processing by the family task and wakes the task if it is sleeping. Queuing, synchronization, and communication mechanisms within the family follow a well-defined set of rules specified by the family.

Regardless of whether the family client called a function synchronously or asynchronously, the accept function always queues the request asynchronously and maintains the set of microkernel message IDs that correspond to messages to which the accept function has not yet replied.

Consider as an example the Open Transport family. It uses the single-task activation model, as shown in Figure 1-6. To receive microkernel messages about client requests, the Open Transport family uses an accept function that executes in the task context of the calling client via the Open Transport client library. Because the calling client task can be preempted by another Open Transport client task making service requests, the accept function must be reentrant. An accept function does not cause a task switch and can access data

within the user and microkernel memory areas directly; thus, it is a very efficient mechanism.

Figure 1-6 Single-task activation model



When an I/O request completes within the Open Transport environment, the Open Transport stream completion notification trickles upstream until it reaches the stream head, and from there the Open Transport family server converts the completion into the appropriate microkernel message ID reply.

By using the single-task model and an accept function, the Open Transport Streams implementation is insulated from the microkernel; it has no knowledge of microkernel structures, IDs, or tasks. On the other hand, the relationship between the accept function and the Open Transport code is complex and asynchronous. The accept function and family server have knowledge of Open Transport data structures and communication mechanisms.

The single-task model is best for families of devices that have either of two characteristics:

- Each I/O request requires little CPU effort. This characteristic applies not only to keyboard and mouse devices but also to direct memory access (DMA) devices, to the extent that the CPU need only set up the transfer.
- No more than one I/O request is ever handled at once. This characteristic might apply to sound, for example, or to any device for which exclusive access is required. It also applies to families that monitor their own scheduling for the interleaving of family I/O processing, such as Open Transport.

Here are the key questions to ask before deciding whether to choose this model:

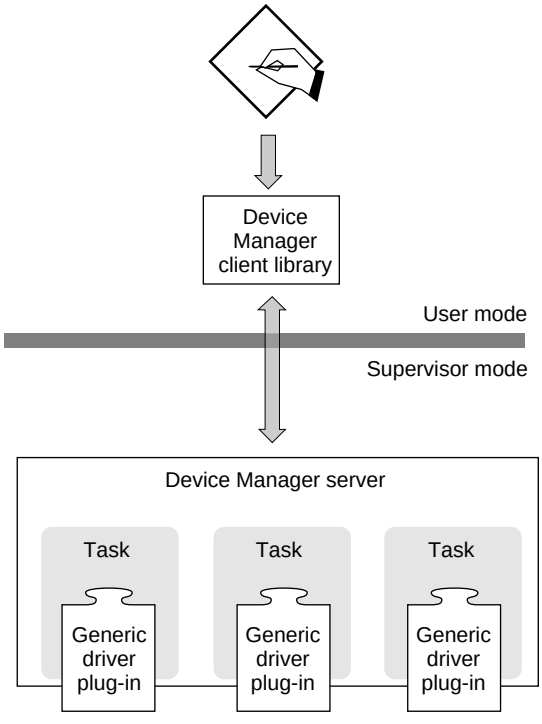
- Can the CPU initiate an I/O request rapidly and then not be involved until the request completes?
- Do supported devices implicitly allow only one I/O request to be completed at a time or, alternately, does the family provide for its own I/O scheduling?

If the answer to either question is yes, the single-task model is the right choice.

Task-per-Plug-in Model

In the task-per-plug-in activation model, for each plug-in instantiated by the family, the family creates a task that provides the context within which the plug-in executes. In Mac OS 8, the Device Manager family uses the task-per-plug-in model, as illustrated in Figure 1-7.

Figure 1-7 Task-per-plug-in model



The server receives requests from calling clients and passes those requests to other family code. The server is responsible for making the data associated with a request available to the family. Typically with this model, the server is implemented as a simple loop that waits for microkernel messages on a message port, or as an accept function. After receiving a message, the server delivers the request to the right plug-in to service that request. In some instances, buffers associated with the original request message may need to be copied or mapped.

When the task associated with a given plug-in gets a request (by whatever mechanisms the family implementation uses), the task calls its plug-in's entry points, waits for the plug-in's response, and then responds to the request.

The plug-in performs the work to actually service the request. It doesn't need to know about the tasking model used by the family or how to respond to

event queues and other family mechanisms. It just needs to know how to perform its particular function.

Device Manager family plug-ins can be concurrent or nonconcurrent. The Device Manager server queues client requests for plug-ins that cannot handle multiple requests concurrently. It makes no subsequent requests to a nonconcurrent plug-in's task until the task signals completion of an earlier I/O request. For concurrent drivers, all queuing and state information describing an I/O request is contained within the plug-in code and data and within any queued requests.

Clients of the Device Manager family can make both synchronous and asynchronous requests. The Device Manager client library makes sure both synchronous and asynchronous clients see appropriate behavior. When a client calls a family function asynchronously, the function causes an asynchronous microkernel message to be sent to the server and then returns to the caller. When a client calls a family function synchronously, the function causes a synchronous microkernel message to be sent to the server and does not return to the caller until the server replies to the message, thus blocking the caller's execution until the I/O request is complete.

Note that when a client is blocked, the plug-in continues to run within its own task context, permitting other clients to make requests of the plug-in while it is processing the first client's synchronous request.

Regardless of whether the client makes a synchronous or an asynchronous request, the behavior of the Device Manager family is the same. For all incoming requests,

- if the target plug-in can handle concurrent requests, the server passes it to the family task associated with the plug-in
- if the target plug-in cannot handle concurrent requests, the server
  - queues the request, if the target plug-in is processing another request
  - passes the request to the family task associated with the plug-in, if the target plug-in is not busy

When a plug-in signals that an I/O operation is complete, the server replies to the original microkernel message. When the client library receives the reply, it either returns to the synchronous client, unblocking its execution, or it notifies the asynchronous client that the I/O is complete.

The task-per-plug-in model is intermediate between the single-task and task-per-request models in terms of the number of tasks it typically uses. It is

best used where the processing of I/O requests varies widely among the plug-ins. In this model, the plug-in runs in a well-defined context and is insulated from microkernel tasking mechanisms and from synchronization issues that result from system resource contention and multiple client requests to a single plug-in.

Task-per-Request Model

The task-per-request model shares the following characteristics with the two activation models already discussed:

- The communication between the family client library and the family server provides the synchronous or asynchronous calling behavior requested by family clients.
- The client library and server use microkernel messages to communicate with each other about I/O requests.

In the task-per-request model, the server's interface to the rest of the family implementation is completely synchronous.

In this model, one or more tasks created by the family, and, optionally, an accept function, wait for messages on the family's message port. An arriving message containing information describing an I/O request awakens one of the tasks, which calls a family function to service the request. All state information necessary to handle the request is maintained in local variables of the task. The task is blocked until the I/O request completes, at which time it replies to the microkernel message to indicate the result of the operation. After replying, the task waits for more messages.

As a consequence of the synchronous nature of the interface between the server and the family implementation, code calling through this interface must be running as a blockable task. This calling code is either the task provided by the family to service the I/O (for asynchronous I/O requests) or the client's task (for synchronous requests received by an accept function).

The task-per-request model is best for a family where an I/O request can require continuous attention from the CPU and multiple I/O requests can be in progress simultaneously. A family that supports simple, high-bandwidth devices is a good candidate for this model. (A simple device lacks built-in intelligence that enables it to support features such as multiple outstanding requests or reordering of requests.) The Mac OS 8 File Manager uses the task-per-request model in processing asynchronous I/O requests.

One problem associated with this activation model is tuning the number of tasks to permit the desired level of concurrence. Tuning can be done dynamically: When the family detects that performance could benefit from more tasks to process more requests concurrently and there are resources to permit it, new tasks can be created as needed. Similarly, when resources become scarce or the number of concurrent requests is much smaller than the number of tasks available to handle them, some tasks can be destroyed, freeing their resources for other uses.

When a family uses the task-per-request model, the family's plug-ins must be reentrant and the family must provide the plug-ins with synchronization services. In addition, the family designer must provide to plug-in developers a set of programming rules and guidelines regarding the correct use of the synchronization services.

Family Programming Issues

The choice of activation model is the biggest family programming issue. Each of the models discussed previously has merit. Within each model, there are issues to be addressed. The single-task and task-per-plug-in models require state information to be stored either within the libraries, the plug-ins, or the family server code, or within some combination of those. The task-per-request model is the simplest model, but it will probably be the most expensive model in terms of system overhead. It makes heavy use of microkernel messages and tasking resources.

Unless there are multiple task switches within a family, the system overhead is identical within all of the activation models. The shortest task path from application to I/O is completely synchronous because all code runs in the context of the caller's task. For a long I/O path, through multiple families, the greater the use of synchronous calls, the smaller the number of task switches. However, using only synchronous calls decreases the responsiveness of the application making the request— its activity stops pending the completion of an outstanding I/O request. Providing at least one level of asynchronous call between an application and an I/O request results in the best responsiveness from the user perspective. Within the file system, a task switch at the File Manager allows a user-visible application, such as the Finder, to continue. The File Manager assigns an I/O request to one of its tasks, and that task might be used via synchronous calls by the block storage and SCSI families to complete their part in I/O transaction processing.

This kind of short-cut communication between families requires a very clear understanding of the relationships between the families, including the stack needs of the called family, the activation model of the called family, and the asynchronous and synchronous paradigms used by the called family. This is part of the decision-making process in developing each family activation model.

Name Registry

The name registry is a centralized, runtime database that stores system hardware and software configuration information. Information stored in the name registry comprises both static and dynamic data maintained by various components of the system. The name registry is organized as a tree-structured collection of entries, each of which can contain an arbitrary number of name-value pairs called *properties*.

The device portion of the name registry describes the configuration and connectivity of the hardware in the system. Each entry in the device subtree has properties that describe the hardware represented by the entry and may contain a reference to the plug-in that controls the device, as well as related software configuration information for families and plug-ins.

The name registry supports important features of the I/O architecture of Mac OS 8, including the following:

- **Effective driver replacement.** This capability allows you to release updates to drivers.
- **Dynamic driver loading and unloading.** The name registry provides a dynamic and flexible environment for identifying devices. This type of capability is necessary for supporting devices such as hot swappable PC cards.
- **Simplification of driver writing.** The name registry provides a consistent way to store, locate, and obtain device-specific information for all devices and device drivers. You won't need to follow different rules for obtaining device-specific information for devices located on the main logic board, a NuBus bus, a PCI bus, a PCMCIA bus, and so forth.
- **Improved portability for device drivers.** The name registry provides the layer of abstraction necessary for driver writers to remove conflicting device

identification and device information callouts (as occurred previously with the Slot Manager) that prevented drivers from being portable to new versions of Macintosh hardware.

Interactions With Experts, DNS, and DFM

During the system booting sequence, low-level family experts describe platform hardware by populating the name registry with device entries. A low-level family expert has specific knowledge of a piece of hardware such as a bus. It knows how physical devices are connected to the system, and it installs and removes that information in the device portion of the name registry. DFM and high-level family experts later peruse this information to locate and select the plug-ins available to the family. When devices are connected to or removed from the system, low-level family experts add and remove information in the device portion of the name registry. In addition, every computer has a *motherboard expert* that understands the main logic board and stores pertinent information about the CPU and memory in the name registry.

Consider a simplified example of how high-level family experts, low-level family experts, the name registry, DNS, and DFM work together to stay aware of dynamic changes in system configuration. Suppose that a Macintosh Duo is docked. The Duo motherboard expert notices that a new SCSI bus and a new video device have appeared within the system. The Duo motherboard expert adds entries for these to the device portion of the name registry and then sends new-device notifications to DNS. The notifications cause DFM to match one or more plug-ins with the devices represented by the new entries. DNS in turn notifies all software that registered to receive new-device notifications (when the new device is a SCSI bus or a video device).

Once notified of the change in the name registry, the SCSI and video family experts scan the plug-ins matched with the new entry and select the one that can best support the new device.

The SCSI expert then probes the new bus for SCSI devices. It adds an entry to the name registry for each SCSI device that it finds. The SCSI expert knows nothing about a particular device for which it adds an entry. Let's suppose it found one disk drive attached to the bus. After adding an entry, the SCSI expert sends a new-device notification, which causes DFM to match one or more plug-ins with the device.The block storage expert gets a notification about the new device, selects and instantiates the best plug-in to manage it, and then creates a name registry entry for a new volume. The File Manager receives

notification of the new volume and notifies the Finder that the volume is available. The volume then appears as an icon on the user's desktop.

I/O Interface

Clients of I/O families do not directly access the name registry. All families provide clients with one or more functions that return information about all the devices or services available through the family. These functions are called *iteration functions*.

A family's iteration function returns an array of data structures called *I/O iterator structures*. Each family's iterator structure has two parts: a part that is common across all families and a part that is family-specific and that describes a given device.

The `IOCommonInfo` data type defines the information that is common to all I/O families. It consists of a device reference number and a version number.

The device reference number uniquely identifies a device within a family, It does not necessarily uniquely identify a device across all I/O families. The version number identifies the version of the family's iterator structure that is in use.

```
struct IODeviceRef {
    UInt32 contents[4]; /* family's unique identifer for a device */
};
typedef struct IODeviceRef IODeviceRef;

typedef UInt32 IteratorDescVersion;     /* version of I/O iterator
                                           structure in use by family */

struct IOCommonInfo {
    IODeviceRef             ref;
    IteratorDescVersion     versionNumber;
};

typedef struct IOCommonInfo IOCommonInfo;
```

Compatibility—Backward and Forward

The following sections discuss compatibility issues for developers of device drivers and applications.

If You Develop Device Drivers

Mac OS 8 and its I/O architecture introduce a new environment for device drivers—one that is fundamentally different from that familiar to developers who have created drivers to run on System 7 and older versions of system software. Although Mac OS 8 places some restrictions on drivers, it greatly increases system stability and protects drivers from application error.

The I/O architecture in System 7 is based on resources of type `'DRVR'` and on the Device Manager. Many different types of software use these mechanisms. Some types are affected by the changes introduced by Mac OS 8, and some are not.

Mac OS 8 employs a more restricted concept of driver software. In the I/O architecture of Mac OS 8, a driver is the native code that controls a physical device or that manages a system service. (Code that controls a virtual device such as a RAM disk may also be considered a driver in Mac OS 8.) Software that controls a physical device or manages a system service is affected by the new I/O architecture in Mac OS 8. Examples of this type of software include

- serial drivers (.AIn, .BOut)
- protocol stacks (.MPP, .IPP)
- network drivers (.ENET, ADEVs, MDEVs)
- video drivers (.Display)
- SCSI interface modules (SIMs)

For backward compatibility, Mac OS 8 supports, through the Device Manager, emulated drivers of type `'DRVR'` that do not touch hardware. An emulated driver (`'DRVR'`), such as a print driver, is not a plug-in. An emulated driver runs in user mode outside the I/O subsystem and it can exist only in the traditional application environment that makes use of the `WaitNextEvent` function or Apple event dispatching mechanisms and that allows full access to the Toolbox.

The I/O subsystem of Mac OS 8 is the first complete implementation of the I/O architecture described in this chapter. A subset of the I/O architecture is implemented to support PCI devices on some Power Macintosh models. The document *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the capabilities provided to driver writers for the first PCI-based Power Macintosh computers. If you write a PCI driver according to the specifications there, PCI cards with ROM-based drivers will work unchanged on subsequent PCI-based hardware platforms running Mac OS 8.

The Mac OS 8 driver environment differs from the System 7 driver environment in several ways:

- Mac OS 8 distinguishes between software that runs in user mode and in supervisor mode. In System 7, drivers run in the same environment as applications in a single address space. In Mac OS 8, drivers run in supervisor mode and have access to the microkernel's protected memory space. Applications can't touch the hardware or the driver code or data directly.
- Drivers are packaged as Code Fragment Manager fragments (shared libraries).
- Distinct execution environments are defined in which different sets of services are available. Because drivers execute in supervisor mode and are not made eligible for execution by the Process Manager, they cannot call Toolbox routines. On the other hand, by executing in supervisor mode, drivers gain a fine granularity of control over devices and overall system responsiveness. Drivers use microkernel, driver, and family service libraries as appropriate. Families and their plug-ins are expected to adhere to the rules appropriate to their execution environment.
- Mac OS 8 employs new tasking and messaging mechanisms that allow prioritizing of I/O processing and that make I/O latency predictable. These mechanisms are the foundation for preemptive multitasking and memory protection.
- Drivers exist as plug-ins to a particular I/O family and must conform to the activation model employed by that family. Therefore, when writing your driver, you need to adhere to the plug-in programming interface and the family's implementation guidelines. An I/O family may provide libraries of commonly needed routines, thus simplifying your development effort.

- Drivers that touch hardware must be written in native PowerPC code. As a result, Mac OS 8 will deliver superior I/O performance. Emulated 68K microprocessor drivers that directly access hardware are not supported.

As a result of these changes, you need to change the way you write a device driver. With the exception of drivers written according to specifications for PCI-based Macintosh computers, System 7 drivers that access hardware will *not* run on Mac OS 8.

The next two sections give more information on the separation of application and device driver interfaces and the packaging of driver software, and they describe benefits that result from these changes.

Separation of Application and Device Driver Interfaces

In System 7, all public programming interfaces are available to all varieties of software. Mac OS 8 distinguishes between programming interfaces available to applications and those available to device drivers. Programming contexts become increasingly specialized in Mac OS 8.

In Mac OS 8, drivers have available to them plug-in programming interfaces specifically tuned to the needs of different types of devices, such as display devices or SCSI devices. The plug-in programming interfaces provide control over operating system facilities such as paging and interrupts. Use of plug-in programming interfaces is essential to your driver's portability in future Mac OS releases. These interfaces are guaranteed to be common across Mac OS releases.

Drivers operate outside the application software context in Mac OS 8. As a result, they do not have access to non-reentrant services available to applications.

Common Packaging of Loadable Software

In Mac OS 8, all drivers are created as Code Fragment Manager (CFM) fragments (shared libraries). Each CFM fragment must export a driver description structure that the system uses to locate, load, and initialize the driver.

Mac OS 8 drivers, therefore, are packaged differently from previous Macintosh device drivers. Because they are CFM fragments, they are allowed to have specific static data storage, and they can be written in a high-level language without assembly-language headers. Each instance of a single driver has

private static data and shares code with every other instance of that driver. A device driver no longer locates its private data by means of a field in a Device Unit Table entry.

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

If You Develop Applications

Adjusting to the architectural shift in the I/O subsystem should be relatively easy for the application developer. For compatibility with System 7 applications, the Mac OS 8 Device Manager supports all of the functions described in the chapter "Device Manager" of *Inside Macintosh: Devices*. However, a smaller set of devices will be available through the Device Manager; for them, the system supports a compatibility layer that converts old function calls to new ones. Thus, if your application calls the Device Manager, it will continue to run on Mac OS 8, but it will incur a performance penalty going through the compatibility layer.

For better performance and for access to services well suited to a given class of devices, you should update your application. Instead of the Device Manager, you should use the programming interface provided by the family to which the device belongs. For example, if your application uses the Display Manager, you benefit from a set of routines tuned to work with display devices.

In many cases, Mac OS 8 interfaces will be the same as or very similar to existing programming interfaces, such as those provided in System 7 by the Display Manager and Open Transport. If your application uses these higher-level programming interfaces, it is insulated from underlying changes in the I/O architecture and device drivers and you shouldn't have to change your application so it works with Mac OS 8.

In addition to benefiting from the more effective services available through Mac OS 8 interfaces, adopting the new interfaces now facilitates subsequent development for versions of the Mac OS beyond Mac OS 8. Programming interfaces that Mac OS 8 maintains for backward compatibility with System 7 may not be available with versions of the Mac OS beyond Mac OS 8. For example, the networking paradigm for the Mac OS is changing to Open Transport. Although Mac OS 8 will support System 7 AppleTalk interfaces, later versions of the Mac OS will not. Versions of the Mac OS beyond Mac OS 8 will require you to use the Open Transport programming interface.

If your application ignores public programming interfaces and instead uses nonstandard methods to access a device, you'll need to change your application. In Mac OS 8, attempts by applications to touch hardware will result in access violations. Devices and drivers are not directly accessible to an application. The only access to their services is through an I/O family's client programming interface or an interface maintained for compatibility.

Device Manager Compatibility

In Mac OS 8, the Device Manager functions described in the chapter "Device Manager" of *Inside Macintosh: Devices* are supported. Drivers that provide their services through the Device Manager belong to the Device Manager family and are called *generic drivers*. The Device Manager functions constitute the programming interface for the Device Manager family. The family has its own activation model and set of services, but it is not tuned to the needs of a given type of device.

Although the Device Manager interface is more limiting than those provided by other family interfaces, the Device Manager family offers a migration path to driver developers who implement the basic changes required by Mac OS 8 without totally converting to the I/O architecture in Mac OS 8.

If no family for a device exists, the Device Manager offers a way to use it in Mac OS 8. Consider, for example, a PCI card that receives data, encrypts it, and sends it back. An encryption family doesn't currently exist. By writing the driver according to the rules for drivers of family type `'ndrv'` described in *Designing PCI Cards and Drivers for Power Macintosh Computers*, the card is supported in Mac OS 8 as a plug-in to the Device Manager family.

To summarize, the Device Manager in Mac OS 8 supports drivers that have been revised to run in Mac OS 8 but that have not taken advantage of the enhanced driver services available through I/O families, or for which no family exists. As a result, the Device Manager family's plug-ins are likely to differ quite a bit among themselves, rather than belonging to a general class of devices such as video monitors. For example, Device Manager family plug-ins may include drivers for instrumentation bus adapters, graphics devices, encryption hardware, and so forth. Typically, plug-ins in the Device Manager family are drivers that talk to hardware, but they can also talk to virtual devices such as a RAM disk or loopback software.

For more information on the Device Manager, see "Device Manager Family" (page 8-3).

Glossary

activation model The set of tasking and communication implementation choices made by a family designer that defines both the implementation of the I/O family software and the environment within which a family's plug-ins execute.

consists of the tasking and communication implementation choices made by the family designer. An activation model defines both the implementation of the family software and the environment within which a family's plug-ins execute.

client Any piece of software—including applications, other I/O families and their plug-ins, server programs, and system software—that requests services from an I/O family through the family's programming interface for clients. Compare **plug-in.**

conglomerate library For I/O plug-ins, a family-specific library containing the family-specific services and the generic system services needed by the plug-in. At link time, a plug-in developer specifies the single conglomerate library provided by the family to which the plug-in belongs.

Driver and Family Matching (DFM) service System software that matches hardware-specific software with the I/O devices available in a given Mac-compatible computer.

expert See **family expert**.

family A collection of software pieces that provide a distinct set of I/O services to the system, such as the SCSI family and its SCSI interface modules (SIMs) or the file systems family and its volume format plug-ins. Often, a family is associated with a set of devices that have similar characteristics, such as display devices or ADB devices.

family expert Code within a family that maintains knowledge of the set of family-controllable devices and plug-ins for a given Mac-compatible system.

family server Family software that receives, processes, and responds to service requests from family clients.

family services library A family-specific code library that implements the programming interface for a family's plug-ins. It can optionally provide other

services to the family's plug-ins, such as routines that help the plug-in manipulate data structures or perform tasks central to the service the family provides.

high-level expert See **high-level family.**

high-level family A family whose expert registers to receive notifications about devices that can be controlled by the family. After receiving a notification, the expert inspects the relevant name registry entry(s) and takes appropriate action, such as selecting a plug-in to manage the new device, thus keeping the set of family plug-ins coordinated with changes in the system's hardware configuration.

I/O plug-in A dynamically loaded piece of software that provides to family clients a particular implementation of the service offered by an I/O family. Within the file systems family, for example, a volume-format plug-in implements file system services for a specific volume format.

low-level expert See **low-level family.**

low-level family A family whose expert has specific knowledge of a piece of hardware, such as a bus or a main logic board. The expert knows how physical devices are connected to the system, and can detect when a device that can be controlled by the family is added or removed. When such events occur, the expert adds or removes information in the device tree portion of the name registry and sends notifications to the Driver and Family Matching service.

plug-in See **I/O plug-in.**

Driver and Family Matching

Contents

The Driver and Family Matching (DFM) service matches hardware-specific software with the I/O devices in the system. This chapter describes the types of devices the DFM service handles, the way it works, and the data structures I/O device plug-ins and family experts must export.

About the Driver and Family Matching Service

The DFM service enables the Mac OS 8 kernel to be significantly hardware independent. It accomplishes this by loading hardware-specific software components, such as device driver plug-ins and family experts, at system initialization time. The hardware-specific software can be stored in ROM, on disk, or in both places. The DFM service is responsible for

- locating the correct hardware-specific software for each device in the system,
- loading plug-in code into memory,
- generating notifications to clients of the device, and
- installing a pointer to the plug-in at the proper place in the Name Registry.

Device Categories

For device and family matching, I/O devices can be categorized into the five groups described in this section.

Simple Device

A simple device is one that matches only a single device-specific plug-in. Although the DFM service may return pointers to more than one version of the plug-in—for example, one in ROM and multiple versions on mass storage media—the family expert can simply select the one on the mass storage media with the highest version number. A SCSI device is a good example of a simple device: every SCSI controller has a controller-specific plug-in.

Multiple-Emulation Devices

Multiple-emulation devices can operate in multiple modes or emulations; they can have a separate plug-in for each emulation. For example, a serial port

controller can use a serial plug-in to communicate with a modem or printer and an AppleTalk plug-in to communicate over the network. The DFM service finds all the relevant plug-ins during the matching process. The family expert then decides which plug-in to load, based on the use to which the device is applied, as specified by the user and stored in a preferences file. If the plug-ins located by the DFM service belong to more than one family, all the relevant family experts are instantiated.

Multiple-Plug-in Devices

Multiple-plug-in devices have more than one associated plug-in, but only one of them provides the best performance. For example, a joystick connected to the Apple Desktop Bus port matches both the Apple mouse plug-in and a vendor-specific plug-in. In this case, the pointing family expert must determine which plug-in is best suited to the device, using family-specific algorithms.

Multifunction Cards

Multifunction cards are devices that support more than one function, such as some NuBus™ and PCMCIA cards. Such a device can have a single plug-in to support all of its functions, or it can have a separate plug-in for each of its functions. Multifunction cards fall into one of the categories described previously, depending on how they are represented in the Name Registry.

Virtual Devices

A virtual device is software that provides I/O capability independently of any specific piece of hardware. Virtual devices are typically associated with high-level families. For example, a disk partition is a virtual device that belongs to the block storage family. The DFM service does not make any distinction between virtual and real devices.

Use of the Name Registry

The Name Registry is a data structure, maintained by the Mac OS, that stores hardware and plug-in configuration information for the I/O subsystem, in addition to other data used by other parts of the system. The Name Registry contains a set of name entries, each of which has an arbitrary-size set of properties. Each property has a name and a value describing configuration information pertinent to the name entry.

During the Mac OS 8 boot sequence, the DFM service imports I/O device data from the device tree provided by the Open Firmware standard booting code. The DFM service places this information into the Mac OS 8 Name Registry, after pruning away information about device drivers belonging to other operating systems. This data creates the hardware subtree within the Name Registry that describes the I/O devices available within the current system configuration.

Name entries are installed in and removed from the Name Registry by low-level expert software whenever devices are connected or disconnected from the system. The DFM service stores and maintains descriptions and pointers to plug-ins in the Name Registry.

Loading Plug-Ins and Family Experts

Generally, applications, plug-ins, and family experts avoid invoking the DFM service directly. Every plug-in and family expert must export a data structure to be used for matching. The DFM service uses these data structures to locate plug-in and family expert code. This mechanism enables the DFM service to load only the required code, thereby reducing the memory footprint of the system.

For users, installing a plug-in involves simply copying the plug-in file into the Hardware Support folder (inside the Mac OS folder) where the DFM service will find it and notify the family. Families are also installed this way. Plug-ins are instantiated multiple times—once for each device to which the plug-in is matched. Although each instance of a plug-in has its own data, only one copy of the plug-in's executable code exists in memory and is shared by all instances. Family experts are instantiated only once.

The DFM service is automatically invoked during the boot sequence. For each name entry in the device portion of the Name Registry, the DFM service locates all the matching plug-ins and family experts using information in the matching data structures exported by each plug-in and family expert. See "Driver and Family Matching Constants and Data Types" (page 2-10) for descriptions of the matching data structures.

After locating all the family experts and plug-ins associated with a device, the DFM creates `driver-ptr` and `driver-description` properties for those with entries in the device subtree of the Name Registry. The `driver-ptr` property is an array of pointers to the available plug-ins. The `driver-description` property is an array of the matching data structures belonging to the available plug-ins.

The DFM service also loads the required family experts into memory, if not previously loaded, and calls their main (initialization) entry point. When called at their initialization entry point, family experts are required to set up their internal data structures and register their service category with the Device Notification Service.

After the boot sequence is complete, the DFM service notifies family experts of every device that they can control whenever such devices are discovered. The family expert is responsible for selecting the most suitable plug-in, instantiating it, and deleting the other plug-ins from memory (unless they are likely to be required in the future). When a family expert is notified about a new device, it scans the driver-description array to find the most suitable plug-in for its device family (also called *service category*). Then the family expert locates the `driver-ptr` property associated with the selected plug-in and invokes the DFM service (which invokes the Code Fragment Manager) to load the plug-in into memory. The DFM service ensures that plug-ins already present in memory are not loaded again.

The DFM service provides a programming interface for family experts to use to unload unnecessary plug-ins. After selecting the set of most suitable plug-ins, the family expert invokes DFM service functions to remove all other plug-ins from memory.

Matching Mechanisms

The DFM service uses the algorithm described in this section to find the matching plug-ins and families for a given device entry in the Name Registry. See "Driver and Family Matching Constants and Data Types" (page 2-10) for descriptions of the fields and structures named in this section.

The DFM service implements two matching mechanisms: a standard mechanism by which a single plug-in is matched with a single device, and a generic mechanism by which a single plug-in can be matched with any of a set of devices.

Standard Matching

For a given Name Registry device entry, the DFM service uses the `nameInfoStr` field of the `DriverType` structure to locate all the plug-ins in the Hardware Support folder with a file type of `'ndrv'`. Once a set of plug-ins is located, the DFM service uses the `serviceCategory` field of the `DriverServiceInfo` structure

to locate the associated families. All of these structures are contained within the `DriverDescriptor` structure.

If no plug-in is found for a given Name Registry entry, the DFM service searches for the matching low-level expert in a predefined folder with a file type of `'expt'`. The DFM service uses the `DeviceName` array in the `FamilyDescriptor` structure to locate the low-level expert.

Generic Matching

The DFM service provides a generic matching mechanism by which a developer can specify a level of compatibility for a plug-in, ranging from generic to specific. A single generic plug-in can drive a set of devices. For example, a disk vendor could develop a single driver for all compatible models or for only a specific revision. Or a third-party developer could develop a plug-in to match hardware produced by several other companies.

To use the generic matching capability of the DFM service, family experts create a `matching` property with multiple values for device entries in the Name Registry, based on the information available when the expert scans for devices. The DFM service matches this information with the `nameInfoStr` value exported by plug-ins. The values of these properties are built from four pieces of information associated with devices:

- manufacturer
- product number
- revision number
- function performed by the device

When a low-level expert discovers a device added to the system after boot time, the low-level expert creates a new entry in the device subtree of the Name Registry with the appropriate `name` and `matching` property values, using the hardware information available from the device. If hardware probing does not provide enough device-specific information, the family expert may need to invoke all matching plug-ins before selecting the most suitable one.

The DFM service can match a single plug-in to a set of products using multiple `matching` property values created according to the conventions described in the following paragraphs. The fields should be separated by the `$` character (ASCII value 36). Numeric values should be expressed by four ASCII characters, left-padded with zeroes, representing hexadecimal notation (for example, the hexadecimal value 0x99 should be represented by the ASCII string `'0099'`).

Name Property

To match a single plug-in to a particular product revision level, families define the device `name` property in the sequences shown in Table 2-1.

Table 2-1 DFM conventions for `name` property value

Family	String sequence for `name` property value
NuBus	<vendor ID or `NULL`>`$`<function category>`$`<function subcategory>`$`<board ID>
PCI or PC	<name created by Open Firmware> or <name exported by hardware device> or <vendor ID (11 bytes maximum)>`$`<product part number (9 bytes maximum)>`$`<product revision (10 bytes maximum)>
SCSI	<vendor ID (11 bytes maximum)>`$`<product part number (9 bytes maximum)>`$`<product revision (10 bytes maximum)>
IDE	<model number>
ADB	`adb$`<bus address>`$`<handler ID>

Matching Property

To match a single plug-in to multiple products, the family defines the values of the `matching` property in the sequences shown in Table 2-2.

Table 2-2 DFM conventions for `matching` property values

Value	Family	String sequence
1*	NuBus, IDE, or ADB	No value
1	PCI, PC, or SCSI	<vendor ID>`$`<product part number>
2†	NuBus, IDE, or ADB	No value
2	PCI, PC, or SCSI	<vendor ID>`$`<function ID>`$`<bus type>
3‡	IDE	No value
3	NuBus	<vendor ID or `NULL`>`$`<function category>`$`<function subcategory>
3	PCI, PC, or SCSI	<vendor ID>`$`<function ID>
3	ADB	`adb$`<bus address>
4§	NuBus or IDE	No value
4	PCI, PC, or SCSI	<vendor ID>
4	ADB	`adb`
5¶	ADB	No value
5	NuBus, PCI, PC, SCSI, or IDE	<function ID>`$`<bus type# or chip set name**>

* Value 1 matches all revisions of a product.
† Value 2 matches products from a single vendor with the same function and bus type.
‡ Value 3 matches products from a single vendor with the same function.
§ Value 4 matches all products from a single vendor.
¶ Value 5 matchesplug-ins that export only one name, which determines how many devices it can match.
\# For block storage devices and card enablers.
** For other devices such as modems and networking devices.

Note

The generic matching mechanism just described works well after low-level family experts are running and have updated the device subtree in the Name Registry with their correct `matching` property. However, boot devices cannot use this approach because their `name` and `matching` properties are created by Open Firmware (the industry-standard booting code used by Mac OS 8), which may not follow these conventions.

Driver and Family Matching Constants and Data Types

Plug-In Description Structure

The plug-in description structure is defined by the `DriverDescription` data type. This structure is equivalent to the driver description structure described in *Designing PCI Cards and Drivers for Power Macintosh Computers,* although some of its fields have been overloaded with interpretations specific to Mac OS 8. The DFM service uses the `DriverDescription` structure to match plug-ins with family experts and set up the plug-in's runtime environment.

```
struct DriverDescription {
    OSType                      driverDescSignature;
    DriverDescVersion           driverDescVersion;
    DriverType                  driverType;
    DriverOSRuntime             driverOSRuntimeInfo;
    DriverOSService             driverServices;
};

typedef struct DriverDescription DriverDescription;
typedef struct DriverDescription *DriverDescriptionPtr;
```

Field Descriptions

`driverDescSignature` Signature of this `DriverDescription` structure. You should supply this field with the value `kDriverDescriptionSignature` as defined in the plug-in description signature enumeration (page 2-11).

`driverDescVersion` Version of this driver description structure. You should supply this field with the value `kVersionOneDriverDescriptor` as enumerated for the `DriverDescVersion` data type (page 2-11).

`driverType` Structure that contains plug-in name and version, described in "Plug-in Type Structure" (page 2-12).

`driverOSRuntimeInfo`
: Structure that contains bit flags specifying options and other information to define the plug-in runtime environment, described in "Plug-in Runtime Structure" (page 2-12).

`driverServices`
: Structure used to declare the plug-in's device family (or service category) and service type (subcategory), described in "Plug-in Services Information Structure" (page 2-15).

Plug-in Description Signature

The plug-in description signature is defined by the following enumerated values, which are used in the `driverDescSignature` field of the plug-in description structure (page 2-10).

```
enum {
    kTheDescriptionSignature    = 'mtej',
    kDriverDescriptionSignature = 'pdes'
};
```

Enumerator descriptions

`kTheDescriptionSignature`
: Plug-in is for Device Manager family (`'ndrv'`).

`kDriverDescriptionSignature`
: Plug-in is Mac OS 8 version.

Plug-in Description Version

The plug-in description version is defined by the `DriverDescVersion` data type and its enumerated values, which are used in the `driverDescVersion` field of the plug-in description structure (page 2-10). The version differentiates plug-in description structures having the same `driverDescSignature` value.

```
typedef UInt32 DriverDescVersion;
enum {
    kInitialDriverDescriptor    = 0,
    kVersionOneDriverDescriptor = 1
};
```

Enumerator descriptions

`kInitialDriverDescriptor`
Plug-in description structure is for Device Manager family (`'ndrv'`).

`kVersionOneDriverDescriptor`
Plug-in description structure is Mac OS 8 version.

Plug-in Type Structure

The `DriverType` data type defines the plug-in type structure, which contains plug-in name and version information used to match the plug-in to a specific device. The `DriverType` structure is used in the `driverType` field of the plug-in description structure (page 2-10).

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

typedef struct          DriverType DriverType;
typedef                 DriverType *DriverTypePtr;
```

Field Descriptions

`nameInfoStr` Name used to identify the plug-in and distinguish among versions of the plug-in when the DFM service is searching for plug-ins. This string of type `Str31` is used to match the `name` and `matching` properties in the Name Registry.

`version` Version number used to obtain the newest plug-in when several identically named plug-ins (that is, plug-ins with the same value of `nameInfoStr`) are available on disk.

Plug-in Runtime Structure

The plug-in runtime structure, represented by the `DriverOSRuntime` data type, contains information used to set up and maintain the plug-in's runtime environment. The plug-in runtime structure is used in the `driverOSRuntimeInfo` field of the plug-in description structure (page 2-10).

```
struct DriverOSRuntime {
    RuntimeOptions          driverRuntime;
    Str31                   driverName;
    UInt32                  driverDescReserved[8];
};

typedef struct DriverOSRuntime DriverOSRuntime;
typedef struct DriverOSRuntime *DriverOSRuntimePtr;
```

Field Descriptions

`driverRuntime` Options used to determine runtime behavior of the plug-in. You can supply this field with one of the values defined in the runtime options enumeration (page 2-13).

`driverName` Driver name used by Mac OS if the plug-in family is `'ndrv'`. This field is unused for other plug-in families.

`driverDescReserved` Reserved for future use.

Runtime Options

Runtime options are defined by the `RunTimeOptions` data type, which is used in the plug-in runtime structure (page 2-12) to specify the runtime behavior of the plug-in. The runtime options are mutually exclusive.

```
typedef OptionBits RuntimeOptions;
enum {
    kDriverIsLoadedUponDiscovery    = 0x00000001,
    kDriverIsOpenedUponLoad         = 0x00000002,
    kDriverIsUnderExpertControl     = 0x00000004,
    kDriverIsConcurrent             = 0x00000008,
    kDriverQueuesIOPB               = 0x00000010,
    kDriverIsLoadedAtBoot           = 0x00000020,
    kDriverIsForVirtualDevice       = 0x00000040
};
```

Enumerator descriptions

`kDriverIsLoadedUponDiscovery` The bit indicating that the family expert loads the plug-in when hardware requiring the plug-in is discovered.

`kDriverIsOpenedUponLoad`
: The bit indicating that the system opens the plug-in when it is loaded. This option is supported for backward compatibility only; it is not used in Mac OS 8.

`kDriverIsUnderExpertControl`
: The bit indicating that the family expert handles plug-in load and open requests. For Mac OS 8 this bit should always be set.

`kDriverIsConcurrent`
: The bit indicating that the plug-in is capable of handling concurrent requests. This option is supported for backward compatibility only; it is not used in Mac OS 8.

`kDriverQueuesIOPB`
: The bit indicating that the Device Manager does not queue the IOPB to the DCE request before calling the plug-in. This option is supported for backward compatibility only; it is not used in Mac OS 8.

`kDriverIsLoadedAtBoot`
: The bit indicating that the plug-in is loaded at boot time.

`kDriverIsForVirtualDevice`
: The bit indicating that the plug-in is for a virtual device. the DFM service will create an entry in the Name Registry for this plug-in.

Plug-in Services Structure

The plug-in services structure, represented by the `DriverOSService` data type, is used in the `driverServices` field of the plug-in description structure (page 2-10). The `DriverOSService` data type describes the families required for this plug-in to work correctly. A plug-in can belong to more than one family, although it is not recommended for plug-ins designed for Mac OS 8. In such cases, however, `nServices` should indicate the number of different families that the plug-in supports.

```
struct DriverOSService {
    ServiceCount        nServices;
    DriverServiceInfo   service[1];
};
```

```
typedef UInt32 ServiceCount;
typedef struct DriverOSService DriverOSService;
typedef DriverOSService *DriverOSServicePtr;
```

Field Descriptions

`nServices` The number of families supported by this plug-in. This field determines the size of the service array that follows.

`service` An array of `DriverServiceInfo` structures that specify the supported family programming interface sets.

Plug-in Services Information Structure

The plug-in services information structure, represented by the `DriverServiceInfo` data type, is used in the plug-in services structure (page 2-14). The plug-in services information structure describes the device family (service category) and service type (subcategory) of the family programming interfaces a plug-in supports.

```
struct DriverServiceInfo {
    OSType              serviceCategory;
    OSType              serviceType;
    NumVersion          serviceVersion;
};

typedef struct DriverServiceInfo DriverServiceInfo;
typedef DriverServiceInfo *DriverServiceInfoPtr;
```

Field Descriptions

`serviceCategory` Specifies the device family supported by the plug-in. You can supply this field with one of the values defined in the family constants enumeration (page 2-16).

`serviceType` The DFM service does not use this field. The `serviceType` field is used by families to fine tune plug-in matching. For example, the keyboard and mouse plug-ins could use this field to specify the kind of hardware (serial or ADB) they support or which ADB commands they support.

The generic native device family (`'ndrv'`) uses this field to specify the specific type of device a plug-in supports, using

values defined in the native device family types enumeration (page 2-18).

`serviceVersion` Version (`'vers'`). The DFM service does not use this field.

Family Constants

The following enumerated values are used to represent known I/O device families in the `serviceCategory` field of the plug-in services information structure (page 2-15).

```
enum {
    kServiceCategoryDisplay         = 'disp',
    kServiceCategoryOpenTransport   = 'otan',
    kServiceCategoryBlockStorage    = 'blok',
    kServiceCategoryNdrvDriver      = 'ndrv',
    kServiceCategoryScsiSIM         = 'scsi',
    kServiceCategoryFileManager     = 'file',
    kServiceCategoryIDE             = 'ide-',
    kServiceCategoryADB             = 'adb-',
    kServiceCategoryPCI             = 'pci-',
    kServiceCategoryPCMCIA          = 'pcmc',
    kServiceCategoryDFM             = 'dfm-',
    kServiceCategoryMotherBoard     = 'mrbd',
    kServiceCategoryKeyboard        = 'kybd',
    kServiceCategoryPointing        = 'poit',
    kServiceCategoryRTC             = 'rtc-',
    kServiceCategoryNVRAM           = 'nram',
    kServiceCategorySound           = 'sond',
    kServiceCategoryPowerMgt        = 'pgmt',
    kServiceCategoryGeneric         = 'genr'
};
```

Enumerator Descriptions

`kServiceCategoryDisplay`
Display Manager family.

`kServiceCategoryOpenTransport`
Open transport family.

`kServiceCategoryBlockStorage`
Block storage family.

`kServiceCategoryNdrvDriver`
Device Manager family.

`kServiceCategoryScsiSIM`
SCSI interface module family.

`kServiceCategoryFileManager`
File systems family.

`kServiceCategoryIDE`
ATA family.

`kServiceCategoryADB`
ADB family.

`kServiceCategoryPCI`
PCI family.

`kServiceCategoryPCMCIA`
PC card family.

`kServiceCategoryDFM`
DFM service.

`kServiceCategoryMotherBoard`
Motherboard family.

`kServiceCategoryKeyboard`
Keyboard family.

`kServiceCategoryPointing`
Pointing family.

`kServiceCategoryRTC`
Real time clock family.

`kServiceCategoryNVRAM`
Nonvolatile RAM family.

`kServiceCategorySound`
Sound family.

`kServiceCategoryPowerMgt`
Power management service.

`kServiceCategoryGeneric`
Reserved.

Device Manager Family Types

The following enumerated values are used to represent known I/O device families in the `serviceVersion` field of the plug-in services information structure (page 2-15) to specify the specific type of device a plug-in supports within the Device Manager family (`'ndrv'`).

```
enum {
    kNdrvTypeIsGeneric              = 'genr',
    kNdrvTypeIsVideo                = 'vido',
    kNdrvTypeIsBlockStorage         = 'blok',
    kNdrvTypeIsNetworking           = 'netw',
    kNdrvTypeIsSerial               = 'serl',
    kNdrvTypeIsSound                = 'sond',
    kNdrvTypeIsBusBridge            = 'brdg'
};
```

Enumerator Descriptions

`kNdrvTypeIsGeneric`
Generic device type.

`kNdrvTypeIsVideo`
Video device type.

`kNdrvTypeIsBlockStorage`
Block storage device type.

`kNdrvTypeIsNetworking`
Networking device type.

`kNdrvTypeIsSerial`
Serial device type.

`kNdrvTypeIsSound`
Sound device type.

`kNdrvTypeIsBusBridge`
Bus bridge device type.

ADB Family Reference

Contents

This chapter describes the **ADB family**, which allows clients to get information about and communicate with hardware devices attached to the Apple Desktop Bus (ADB).

Mac OS 8 contains standard keyboard and mouse-handling functions that automatically take care of all required ADB access operations. Most applications typically receive keyboard and mouse input by calling the Apple Event Manager, not by calling the ADB family. For complete information about receiving and interpreting keyboard and mouse input, see *Apple Events in Mac OS 8*.

The ADB family presents ADB services to **ADB clients**, software such as the keyboard and pointing families as well as indirectly to applications that run in user space and to **ADB plug-ins** software modules, also called drivers, for specific families of computers, such as the 6100, 7100, 8100; the 7500, 8500, 9500; or the Powerbook 5300. Whereas the ADB family provides services to clients, the ADB plug-ins actually implement requests for services. For more information on the pointing family, see "Pointing Family Reference" (page 4-5).

About the ADB Family

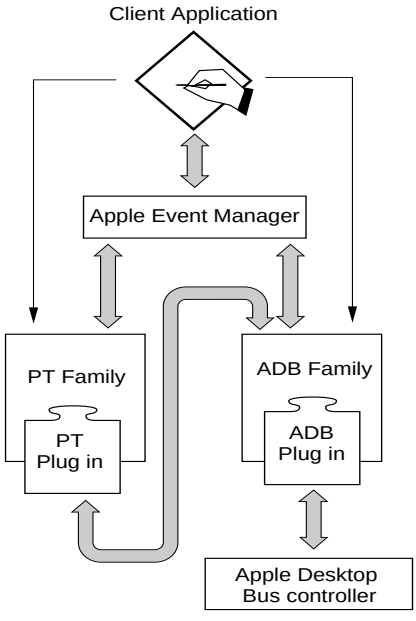
The **Apple Desktop Bus** (**ADB**) is an open-collector, low-speed serial bus that connects user-input devices such as keyboards, mice, graphics tablets, and joysticks to the **Apple Desktop Bus Controller**, the microcontroller that resides in a host computer or in other hardware equipment. Macintosh computers come equipped with one or two ADB connectors. Although a particular model might include two ADB connectors, all models come with only one ADB, which is Apple Computer's standard interface for input peripherals such as keyboards and mouse devices.

An **ADB device** is any peripheral that can connect to the ADB and meets the design requirements described in the "Apple Desktop Bus Controller" chapter of *Macintosh Technology in the Common Hardware Reference Platform*, published by Morgan Kaufman. If you are planning on implementing an ADB device, you should also read the *Apple Desktop Bus Specification* and the Macintosh technical note *ADB–The Untold Story: Space Aliens Ate My Mouse*.

Figure 3-1 illustrates how the ADB programming interface, the ADB server, and the ADB plug-ins allows clients to get information about ADB devices.

- If you need to write a plug-in for your input device or if you are a manufacturer of an ADB peripheral, you can call the ADB family programming interface described in "ADB Client Functions" (page 3-19). (All the ADB client functions begin with the prefix *ADB*.)
- Typically, an application does not need the services provided by the ADB family. However, if you are an application that talks directly to an ADB device, you may call the pointing family programming interface, described in "Pointing Family Reference" (page 4-5) as well as the ADB family programming interface, described in "ADB Client Functions" (page 3-19).
- If you are designing a new computer that has a different Apple Desktop Bus Controller, you need to write a new ADB plug-in that communicates directly with that hardware and to implement the ADB plug-in programming interface described in "ADB Plug-in Defined Functions" (page 3-41). (All the ADB plug-in-defined functions begin with the prefix *MyADBPlugin*.) The ADB family calls the plug-in-defined functions. ADB plug-ins use the ADB family programming interface described in "Functions Exported by ADB Family" (page 3-38). These functions have been implemented by the ADB family for ADB plug-ins to communicate with the family. (All the functions exported by the ADB family begin with the prefix *ADBFam*.)

Figure 3-1 The ADB Family, Its Clients, and Plug-ins



Note
In subsequent developer releases of Mac OS 8, the keyboard and pointing families will most likely become part of an input devices family. ◆

IMPORTANT
Apple Computer, Inc. owns patents on the Apple Desktop Bus. If you want to manufacture a device that works with the ADB software, you must obtain a license and device handler ID from Apple Computer, Inc. Write to this address:

Apple Software Licensing
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014

A license includes a copy of the *Apple Desktop Bus Specification*. ▲

ADB Client Constants and Data Types

This section describes the data types and constants in the ADB family's client programming interface. A client uses the services of the ADB family and its plug-ins to manage data generated by ADB devices.

ADB Connection ID

All the functions in the ADB family programming interface except `ADBResetBus` (page 3-37) require an ADB connection ID. An **ADB connection** is a logical path to an ADB device and serves to control access to the device. Clients can obtain an ID for an ADB connection by calling the `ADBOpen` function (page 3-21).

Clients pass an ADB connection ID to close an ADB connection with the `ADBClose` function (page 3-23) and to retrieve the next autopoll event with the `ADBGetNextAutopoll` function (page 3-34).

Clients also pass an ADB connection ID to obtain and set

- the contents of an ADB register with the `ADBGetRegister` (page 3-24) and `ADBSetRegister` (page 3-26) functions. For more on the ADB registers, see "ADB Register Contents" (page 3-9) and "Getting and Setting the ADB Registers" (page 3-23).
- a handler ID for an ADB connection with the `ADBGetHandlerID` (page 3-28) and `ADBSetHandlerID` (page 3-29) functions respectively. For details on handler IDs, see "Getting and Setting Handler IDs" (page 3-27).
- the status bits for ADB register 3 with the `ADBGetStatusBits` (page 3-31) and `ADBSetStatusBits` (page 3-33) functions respectively. For an illustration of the status bits for ADB register 3, see Figure 3-2 (page 3-31).

Furthermore, clients pass the ADB connection ID to flush an ADB device using the `ADBFlush` function (page 3-36).

The ADB family defines the `ADBConnectionID` data type, an unsigned 32-bit integer that identifies an ADB connection.

```
typedef UInt32 ADBConnectionID;
```

Note
The ADB connection type will probably change in future developer releases of Mac OS 8. ◆

ADB Register Contents

Each device connected to the Apple Desktop Bus may provide up to four registers for storing data. These registers are referred to as **ADB device registers**. An ADB device can implement these registers as it chooses; that is, an ADB register does not have to correspond to an actual hardware register on the ADB device. Clients gain access to an ADB device over the ADB by reading from or writing to these registers. Each ADB device register may store up to 8 bytes of data.

The ADB family defines the `ADBRegisterContents` data type to provide information about the contents of an ADB register. The ADB Register contents data structure is used in the ADB client functions `ADBSetRegister` (page 3-26), `ADBGetRegister` (page 3-24), and `ADBGetNextAutopoll` (page 3-34).

```
typedef struct ADBRegisterContents ADBRegisterContents;

struct ADBRegisterContents {
    Byte            data[8];    /* ADB register data */
    ByteCount       length;     /* ADB register length */
};
```

Field descriptions

`data` Up to 8 bytes of data contained in the ADB register. This value specifies the information in the ADB register.

`length` A byte count that specifies the number of bytes of the `data` field that are valid.

ADB I/O Iterator Data

The ADB I/O iterator data structure provides a device reference, a structure version number, a default address, and a default handler ID for each of the peripherals present. The ADB I/O iterator data structure is defined by the `ADBIOIteratorData` type.

```
struct ADBIOIteratorData {
    IOCommonInfo    IOCI;               /* I/O common information
                                           structure */
    Byte            currentAddress      /* current address of device */
    Byte            defaultAddress;     /* default addresses */
    Byte            defaultHandlerID;   /* default handler IDs */
};

typedef struct ADBIOIteratorData ADBIOIteratorData;
```

Field descriptions

`IOCI`	An I/O common information structure, which defined by the `IOCommonInfo` data type. This structure contains the I/O device reference in its `contents` field. For more on the I/O Common Information data type and the I/O device reference, see "About the I/O Architecture" (page 1-3)
`currentAddress`	A byte that contains the current address of the device.
`defaultAddress`	A byte that contains the default address for the current device referred to by this structure.
`defaultHandlerID`	A byte that contains the default handler ID for the device referred to by this structure (for example, the address the device has on power-up and bus reset).

ADB Match Strings

If you are writing a plug-in for an ADB device that plugs into a higher layer family, such as the pointing family, you'll need to use ADB match strings. All modular I/O families must define match strings for use by the plug-in description data structure. For details on this data structure, also called the driver description data structure, see "Driver and Family Matching" (page 2-3)

For instance, ADB search strings would appear in the driver description data structure of a pointing family plug-in. (For more on the pointing family, see "Pointing Family Reference" (page 4-5).) The order of ADB search strings must always be from the most to the least specific as follows:

3. `ADB-X-YY` where `X = the` default address of the device, and `YY = its` default handler ID. Both strings appear in uppercase hexadecimal.
4. `ADB-X` where `X = the` default address of the device.

ADB Plug-in-Defined Data Types

This section describes constants and data types in the ADB family's programming interface for its plug-ins.

ADB Plug-In Dispatch Table

Each ADB family plug-in must export an ADB plug-in dispatch table, so the ADB family can find the functions it contains. The ADB family calls the Driver and Family Matching Software (DFM) to load each plug-in. For more on DFM, see "Driver and Family Matching" (page 2-3). Subsequently, the DFM returns a pointer to the plug-in dispatch table.

The ADB plug-in dispatch table is defined by the `ADBPluginDispatchTable` data type.

```
struct ADBPluginDispatchTable {
    ADBPluginHeader                 header;             /* header */
    ADBPluginValidateHardwarePtr    ValidateHardware    /* validate hardware */
    ADBPluginInitProc               Init;               /* initialize function*/
    ADBPluginSetAutopollDelayProc   SetAutopollDelay;   /* set autopoll delay */
    ADBPluginGetAutopollDelayProc   GetAutopollDelay;   /* get autopoll delay */
    ADBPluginSetAutopollListProc    SetAutopollList;    /* set autopoll list */
    ADBPluginGetAutopollListProc    GetAutopollList;    /* get autopoll list */
    ADBPluginAutopollEnableProc     AutopollEnable;     /* autopoll enable function */
    ADBPluginAutopollDisableProc    AutopollDisable;    /* autopoll disable function */
    ADBPluginResetBusProc           ResetBus;           /* reset bus function */
    ADBPluginFlushProc              Flush;              /* flush function */
    ADBPluginSetRegisterProc        SetRegister;        /* set register function */
    ADBPluginGetRegisterProc        GetRegister;        /* get register function */
    ADBPluginSetKeyboardListProc    SetKeyboardList;    /* get register function */
};

typedef struct ADBPluginDispatchTable ADBPluginDispatchTable;
```

Field descriptions

`header`	The ADB plug-in header, defined by the `ADBPluginHeader` data type (page 3-12).
`ValidateHardware`	The ADB plug-in defined validate hardware function (page 3-41).
`Init`	The ADB plug-in defined initialization function (page 3-42).
`SetAutopollDelay`	The ADB plug-in defined set autopoll delay function (page 3-43).
`GetAutopollDelay`	The ADB plug-in defined get autopoll delay function (page 3-44).
`SetAutopollList`	The ADB plug-in defined set autopoll list function (page 3-45).
`GetAutopollList`	The ADB plug-in defined get autopoll list function (page 3-46).
`AutopollEnable`	The ADB plug-in defined autopoll enable function (page 3-47).
`AutopollDisable`	The ADB plug-in defined autopoll disable function (page 3-47).
`ResetBus`	The ADB plug-in defined reset bus function (page 3-48).
`Flush`	The ADB plug-in defined flush function (page 3-48).
`SetRegister`	The ADB plug-in defined set register function (page 3-49).
`GetRegister`	The ADB plug-in defined get register function (page 3-50).
`SetKeyboardList`	The ADB plug-in defined set keyboard list function (page 3-51).

ADB Plug-In Header

The ADB plug-in header is defined by the `ADBPluginHeader` data type. Plug-ins use the ADB plug-in header in the `header` field of the ADB plug-in dispatch table (page 3-11).

```
struct ADBPluginHeader {
    UInt32       version;       /* version number formatted like a number
                                   version */
    UInt32      reserved1;      /* reserved for use by Apple */
```

```
    UInt32      reserved2;      /* reserved for use by Apple */
    UInt32      reserved3;      /* reserved for use by Apple */
};

typedef struct ADBPluginHeader ADBPluginHeader;
```

Field descriptions

`version` An unsigned 32-bit integer that specifies the version of the ADB plug-in header. Set this field to the enumerator `kADBPluginCurrentVersion` (page 3-13).

`reserved1` Reserved.

`reserved2` Reserved.

`reserved3` Reserved.

ADB Plug-in Version

The ADB plug-in version enumerator describes the version number of a specific ADB plug-in. The version number appears in the `version` field of the ADB plug-in header data structure, which is defined by the `ADBPluginHeader` data type (page 3-12).

```
enum {
    kADBPluginCurrentVersion    /* ADB plug-in version enumerator*/
};
```

ADB Plug-in Defined-Function Types

This section describes the function pointer types defined by the ADB plug-in programming interface.

ADBPluginValidateHardwareProc

Before the ADB family calls a plug-in's init function, the ADB family calls the validate hardware function provided by the plug-in. The plug-in determines whether the I/O device reference is the device expected by the plug-in.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBValidateHardwareProc) (IODeviceRef *device,
                    Boolean *isMyDevice);
```

For information about creating your own validate hardware function, see the description of the `MyADBValidateHardwareProc` function (page 3-41).

ADBPluginInitProc

When the ADB family selects an ADB plug-in, it calls the initialization function provided by the plug-in. The plug-in then performs appropriate initialization.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginInitProc) (IODeviceRef *device);
```

For information about creating your own initialization function, see the description of the `MyADBPluginInitProc` function (page 3-42).

ADBPluginSetAutopollDelayProc

When the ADB family wants to set the interval between autopoll operations (that is, the **autopoll delay**), it calls the set autopoll delay function provided by the plug-in. The plug-in sets the delay in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginSetAutopollDelayProc)(Duration delay);
```

For information about creating your own set autopoll delay function, see the description of the `MyADBPluginSetAutopollDelayProc` (page 3-43).

ADBPluginGetAutopollDelayProc

When the ADB family wants to find out the autopoll delay (that is, the current interval between autopoll operations), it calls the get autopoll delay function provided by the plug-in. The plug-in then retrieves the autopoll delay in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginGetAutopollDelayProc)(Duration *delay);
```

For information about creating your own set autopoll delay function, see the description of the `MyADBPluginGetAutopollDelayProc` (page 3-44).

ADBPluginSetAutopollListProc

An **autopoll list** is a group of addresses polled during the autopoll mechanism. (These addresses are the group to poll first when trying to clear a service request.) Typically, the autopoll list consists of all the devices that have been opened using the `ADBOpen` function (page 3-21). For more detailed information on autopoll lists, see *Macintosh Technology in the Common Hardware Reference Platform*.

When the ADB family wants to set all the entries in the autopoll list, it calls the set autopoll list function provided by the plug-in. The plug-in then sets the autopoll list in a device-specific manner.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginSetAutopollListProc) (UInt16 addressMask);
```

For information about creating your own set autopoll list function, see the description of the `MyADBPluginSetAutopollListProc` (page 3-45).

ADBPluginGetAutopollListProc

When the ADB family wants to obtain the group of addresses polled during the autopoll mechanism, it calls the get autopoll list function provided by the plug-in. Typically, the autopoll list consists of all the devices that have been opened using the `ADBOpen` function (page 3-21).The plug-in then retrieves the autopoll list in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginGetAutopollListProc)(UInt16 *addressMask);
```

For information about creating your own get autopoll list function, see the description of the `MyADBPluginGetAutopollListProc` function (page 3-46).

ADBPluginAutopollEnableProc

When the ADB family wants to enable autopolling, it calls the autopoll enable function provided by the plug-in. The plug-in then translates this autopoll enabling request in a device-specific fashion. A plug-in must not call the `ADBFamAutopollArrived` function (page 3-40) until the plug-in's autopoll enable function has been called.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginAutopollEnableProc)(void);
```

For information about creating your own autopoll enable function, see the description of the `MyADBPluginAutopollEnableProc` function (page 3-47).

ADBPluginAutopollDisableProc

When the ADB family wants to disable autopolling (that is, turn off the hardware interrupt on autopoll operations), it calls the autopoll disable function provided by the plug-in. The plug-in then translates this autopoll

disabling request in a device-specific fashion. While autopolling is disabled, the `ADBFamAutopollArrived` function (page 3-40) must not be called.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginAutopollDisableProc)(void);
```

For information about creating your own autopoll disable function, see the description of the `MyADBPluginAutopollDisableProc` function (page 3-47).

ADBPluginResetBusProc

When the ADB family wants to send a reset signal over the bus, it calls the reset bus function provided by the plug-in. (The reset bus function is equivalent to a power-up reset.) The plug-in then translates the reset signal request in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginResetBusProc)(void);
```

For information about creating your own reset bus function, see the description of the `MyADBPluginResetBusProc` function (page 3-48).

ADBPluginFlushProc

When the ADB family wants to send a flush command, it calls the flush function provided by the plug-in. The plug-in then translates the command in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginFlushProc)(Byte deviceAddress);
```

For information about creating your own flush function, see the description of the `MyADBPluginFlushProc` function (page 3-48).

ADBPluginSetRegisterProc

When the ADB family wants to set the contents of any of the ADB registers, it calls the set register function provided by the plug-in. The plug-in then translates the request in a device-specific manner.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginSetRegisterProc) (Byte deviceAddress,
                    Byte registerNumber,
                    const ADBRegisterContents *contents);
```

For information about creating your own set register function, see the description of the `MyADBPluginSetRegisterProc` function (page 3-49).

ADBPluginGetRegisterProc

When the ADB family wants to obtain the contents of any of the ADB registers, it calls the get register function provided by the plug-in. The plug-in then translates the command in a device-specific fashion.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginGetRegisterProc) (Byte deviceAddress,
                    Byte registerNumber,
                    ADBRegisterContents *contents);
```

For information about creating your own get register function, see the description of the `MyADBPluginGetRegisterProc` function (page 3-50).

ADBPluginSetKeyboardListProc

When the ADB family wants to tell the plug-in the addresses that have keyboards, called the **ADB keyboard list**, so that it can detect command-power and command-control-power, it calls the set keyboard list function provided by the plug-in. The plug-in then translates the command in a device-specific fashion. For more information about keyboard-specific support, see the chapter

"Apple Desktop Bus Controller" in the document *Macintosh Technology In the Common Hardware Reference Platform*.

The function pointer is defined by the ADB family as follows:

```
typedef OSStatus (*ADBPluginSetKeyboardListProc) (UInt16 addressMask);
```

For information about creating your own set keyboard list function, see the description of the `MyADBPluginSetKeyboardListProc` function (page 3-51).

ADB Client Functions

This section describes the functions used by ADB family clients. Typical clients use the ADB family functions to perform the following actions:

- obtain data about an ADB device including its default address and default handler ID using the `ADBGetDeviceData` function (page 3-20)
- open an ADB connection via the `ADBOpen` function (page 3-21)
- close the ADB connection with the `ADBClose` function (page 3-23)
- obtain notification of an autopoll event by calling the `ADBGetNextAutopoll` function (page 3-34) each time they are waiting for the user to perform an action
- obtain the contents of any of the ADB registers using the `ADBGetRegister` function (page 3-24)
- set the contents of ADB registers 0, 1, and 2 via the `ADBSetRegister` function (page 3-26)
- get or change the handler ID fields of register 3 via the `ADBGetHandlerID` (page 3-28) and `ADBSetHandlerID` (page 3-29) functions
- get or change the status bit fields of register 3 using the `ADBGetStatusBits` (page 3-31) and `ADBSetStatusBits` (page 3-33) functions
- send a flush command over the bus with the `ADBFlush` function (page 3-36)
- send a reset command over the bus using the `ADBResetBus` function (page 3-37)

Getting Information about ADB Devices

Before opening an ADB connection to a device, clients need to obtain data about devices including data size, default address, and default handler ID. The **ADB device address is a** 4-bit bus address that identifies devices of the same type.The **ADB handler ID** is an ADB device-specific 8-bit value. Taken together, a device's default address and handler ID uniquely identify the particular data protocol the device uses for communication.

ADBGetDeviceData

Obtains data about all ADB devices known to the ADB family. Such data includes data size, default address, and default handler ID information.

```
OSStatus ADBGetDeviceData(
                    ItemCount requestCount,
                    ItemCount *totalCount,
                    ADBIOIteratorData *deviceData );
```

`requestCount` An item count that indicates the number of iterator structs in the array pointed to by the `deviceData` parameter.

`totalCount` A pointer to an item count. On output, the `ADBGetDeviceData` function indicates how many ADB devices there are.

`deviceData` On input, a pointer to an array or empty iterator structs. On output, the `ADBGetDeviceData` function fills in the fields for each device it finds up to the number specified in the `requestCount` parameter. For more on the ADB I/O iterator data structure, defined by the `ADBIOIteratorData` type, see "ADB I/O Iterator Data" (page 3-9).

function result An operating system status code. See "ADB Result Codes" (page 3-52) for a list of result codes the ADB family can return.

DISCUSSION

Since there can be a maximum of 16 ADB devices, the `deviceData` parameter should be allocated as 16 iterator structs. In this way, if you pass in the

requestCount parameter as 16, you are guaranteed to obtain all the information you need with one call.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The ADBGetDeviceData function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For a discussion of default device addresses and default handler IDs, see "Getting Information about ADB Devices" (page 3-20).

Opening and Closing an ADB Connection

In order to open an ADB connection, clients need to use the ADBOpen function (page 3-21) to obtain an ADB connection ID. This ADB connection ID creates a logical path to an ADB device specified via an I/O device reference and serves to control access to the device. Once the connection has been opened, the ADB family buffers the data from a small number of autopoll events. To close the connection, clients can use the ADBClose function (page 3-23).

ADBOpen

Opens an ADB connection.

```
OSStatus ADBOpen (const IODeviceRef *ref,
                  ADBConnectionID *connection);
```

`ref`	On input, a pointer to the I/O device reference, defined by the `IODeviceRef` data type, of the device whose connection you want to open. (For more on I/O device references, see "About the I/O Architecture" (page 1-3).) Clients obtain this reference via the `ADBGetDeviceData` function (page 3-20).
`connection`	A pointer to an ADB connection ID. On output, the `ADBOpen` function provides the new connection ID. This value of type `ADBConnectionID` (page 3-8) identifies the ADB connection that the ADB family has opened.
function result	An operating system status code. Your request to open an ADB connection can fail if the connection is already open or if the I/O device ID is invalid. If the connection is already open, `ADBOpen` returns the `adbDeviceBusyErr` result code. If the I/O device ID is incorrectly specified, `ADBOpen` returns the `paramErr` result code. See "ADB Result Codes" (page 3-52) for a list of result codes the ADB family can return.

DISCUSSION

Once the ADB connection has been opened, the ADB family buffers the data from a small number of autopoll events.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBOpen` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To close an ADB connection, you use the `ADBClose` function (page 3-23).

ADBClose

Closes an ADB connection.

```
OSStatus ADBClose (ADBConnectionID connection);
```

`connection` A pointer to an ADB connection ID. This value of type `ADBConnectionID` (page 3-8) identifies the ADB connection you want the ADB family to close. Clients obtain this connection ID via the `ADBOpen` function (page 3-21).

function result An operating system status code. Your request to close an ADB connection can fail if the connection has already been closed. If you incorrectly specify the connection ID of the ADB connection, `ADBClose` returns the `adbInvalidConnectionIDErr` result code. See "ADB Result Codes" (page 3-52) for a list of the result codes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBClose` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To open an ADB connection, you use the `ADBOpen` function (page 3-21).

Getting and Setting the ADB Registers

An ADB device has four ADB logical registers each of which contains up to 8 bytes of data. Clients can get or set the contents of any one register at a time.

- Register 0 contains the autopoll data. For example, on a mouse, register 0 contains the relative offset since the last time it was read.
- Register 1 and 2 are device-specific.
- Known as the **ADB control register**, register 3 is two bytes long. It stores the device address, handler ID, and four **ADB status bits**. The *ADB Specification* describes the protocol for changing these fields. All ADB devices must follow this protocol to ensure that dynamic address assignment and address resolution work.

The `ADBGetRegister` function works on all four registers (0, 1, 2, and 3). The `ADBSetRegister` function only works on ADB registers 0, 1, and 2.

To change the contents of register 3, you need to use the ADB family functions that get and set its individual fields. The `ADBGetHandlerID` and `ADBSetHandlerID` functions, described in "Getting and Setting Handler IDs" (page 3-27) work only on register 3 as do the `ADBGetStatusBits` and `ADBSetStatusBits` functions described in "Getting and Setting ADB Status Bits" (page 3-30).

ADBGetRegister

Retrieves the contents of an ADB register.

```
OSStatus ADBGetRegister (
                    ADBConnectionID connection,
                    Byte registerNumber,
                    ADBRegisterContents *contents,
                    AbsoluteTime *timestamp);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose register contents you want to retrieve.

`registerNumber`
A byte that you set to describe the register (0, 1, 2, or 3) whose contents you want to obtain.

`contents` A pointer to a structure of type `ADBRegisterContents` (page 3-9). On output, the `ADBGetRegister` function sets the structure to the contents of the register specified by the `registerNumber` parameter.

`timestamp`	A pointer to an absolute time value. On output, the `ADBGetRegister` function specifies when the contents of the register were retrieved taken at hardware interrupt time.
function result	An operating system status code. If you have specified a register number incorrectly, `ADBGetRegister` returns the `paramErr` result code. If you have specified an ADB connection ID incorrectly, the function returns the `adbInvalidConnectionIDErr` result code. If the specified device doesn't respond, the function returns the `adbDeviceTimeoutErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

Clients who are familiar with the System 7 Talk command, defined in the *ADB Specification* may want to know that a call to the `ADBGetRegister` function is the equivalent of sending a **Talk command** over the wire. A Talk command fetches user input or other data from an ADB device and requests that the specified device send the contents of a specified register over the bus.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBGetRegister` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set ADB register 0, 1, or 2, you use the `ADBSetRegister` function (page 3-26). For an overview of the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23). To set the contents of register 3, you need to use the ADB client functions for changing the individual fields described in "Getting

and Setting Handler IDs" (page 3-27) and "Getting and Setting ADB Status Bits" (page 3-30).

ADBSetRegister

Sets the contents of ADB registers 0, 1, or 2.

```
OSStatus ADBSetRegister (
                    ADBConnectionID connection,
                    Byte registerNumber,
                    const ADBRegisterContents *contents);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose register contents you want to retrieve.

`registerNumber` A byte that you set to describes the register (0, 1, or 2) whose contents you want to change.

`contents` A pointer to a structure of type `ADBRegisterContents` (page 3-9). On output, the `ADBSetRegister` function fills in this structure to describe the contents of the ADB register specified in the `registerNumber` parameter as well as its `length` and `data` fields.

function result An operating system status code. If you have specified a register number incorrectly, `ADBSetRegister` returns the `paramErr` status code. If you have specified the ADB connection ID incorrectly, the function returns the `adbInvalidConnectionIDErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

Those clients who are familiar with the System 7 Listen command may want to know that a call to `ADBSetRegister` is the equivalent of sending a **Listen command** over the wire. A Listen command (defined in the *ADB Specification*) instructs a device to prepare to receive additional data.

Note
`ADBSetRegister` only sets registers 0, 1, and 2. Use the functions described in "Getting and Setting Handler IDs" (page 3-27) and "Getting and Setting ADB Status Bits" (page 3-30) to set individual fields of register 3. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBSetRegister` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To retrieve the contents of any ADB register, you use the `ADBGetRegister` function (page 3-24). For an overview of the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23).

To set the individual fields of the contents of register 3, you can use the `ADBSetHandlerID` function (page 3-29) and the `ADBSetStatusBits` function (page 3-33).

Getting and Setting Handler IDs

The **ADB handler ID** is an ADB device-specific 8-bit value. Taken together, an ADB device's default address and handler ID uniquely identify the particular data protocol the device uses for communication. Devices may support more than one protocol. Changing a device's handler ID may change its protocol.

The handler ID for a device occupies the second byte of ADB register 3. You can use the `ADBGetHandlerID` (page 3-28) and `ADBSetHandlerID` (page 3-29) functions, respectively, to obtain and set the handler IDs.

For an illustration of the contents of device register 3, see Figure 3-2 (page 3-31).

Note
The upper 4 bits of register 3 contain the status bits. To get and set these bits, you use the `ADBGetStatusBits` and `ADBSetStatusBits` functions described in "Getting and Setting ADB Status Bits" (page 3-30). The next 4 bits contain the device's address, which cannot be changed via a programming interface. ◆

ADBGetHandlerID

Obtains a handler ID for an ADB connection.

```
OSStatus ADBGetHandlerID (
                    ADBConnectionID connection,
                    Byte *handlerID);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose handler ID you wish to retrieve.

`handlerID` A pointer to a byte. On output, the `ADBGetHandlerID` function uses this byte to describe the handler ID for ADB register 3.

function result An operating system status code. If you specify an incorrect connection ID, `ADBGetHandlerID` returns the `adbInvalidConnectionIDErr` status code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBGetHandlerID` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To retrieve the contents of any ADB register, you use the `ADBGetRegister` function (page 3-24). To set the contents of ADB registers 0, 1, and 2, you use the `ADBSetRegister` function (page 3-26). For an overview of the ADB registers, see "Flushing the ADB" (page 3-36).

To set the handler ID for an ADB register, you can use the `ADBSetHandlerID` function (page 3-29). To set the `ADBSetStatusBits` function (page 3-33). For a discussion of handler IDs, see "Getting and Setting Handler IDs" (page 3-27).

ADBSetHandlerID

Sets the handler ID for an ADB connection.

```
OSStatus ADBSetHandlerID (
                    ADBConnectionID connection,
                    Byte handlerID);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose handler ID you wish to retrieve.

`handlerID` A byte that describes the handler ID for ADB register 3.

function result An operating system status code. Some handler IDs (for instance `00`, `FF`, `FE`, and `FD`) are reserved by the *ADB Specification* for special functions. `ADBSetHandlerID` returns the `adbReservedHandlerIDErr` result code if it is unable to set the handler ID specified. If the device does not support the handler ID you have specified, `ADBSetHandlerID` returns the `adbInvalidHandlerIDErr` result code. If you have incorrectly specified the ADB connection ID, the function returns the `adbInvalidConnectionIDErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

`ADBSetHandlerID` calls the `ADBGetHandlerID` function (page 3-28) to verify that the handler ID you have specified is supported.

IMPORTANT
Changing a device's handler ID may change that device's protocol. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBSetHandlerID` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To retrieve the contents of any ADB register, you use the `ADBGetRegister` function (page 3-24). To set the contents of ADB registers 0, 1, and 2, you use the `ADBSetRegister` function (page 3-26). For an overview of the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23).

To obtain the handler ID for an ADB register, you can use the `ADGetHandlerID` function (page 3-28). For a detailed discussion of handler IDs, see "Getting and Setting Handler IDs" (page 3-27).

To set the status bits in register 3, you can use the `ADBSetStatusBits` function (page 3-33). To retrieve the status bits in register 3, you can use the `ADBGetStatusBits` function (page 3-31). For a detailed discussion of status bits, see "Getting and Setting ADB Status Bits" (page 3-30)

Getting and Setting ADB Status Bits

The `ADBGetStatusBits` and `ADBSetStatusBits` functions get and set the **status bits,** which are the 4 most significant bits of the control register (also called ADB register 3).

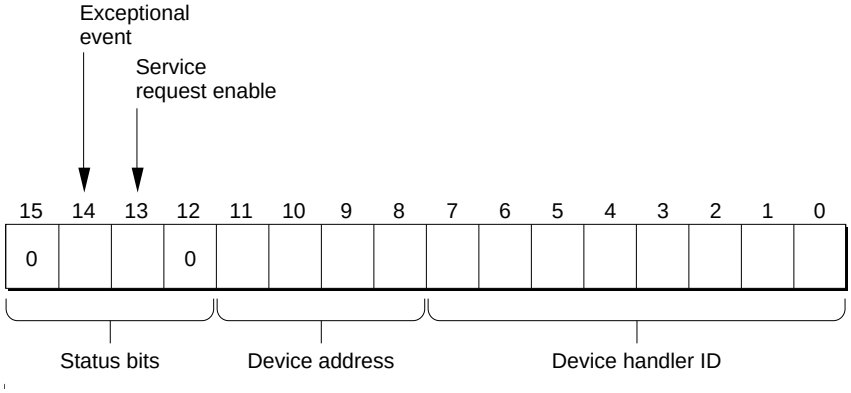
The upper four bits of ADB register 3 contain

- a reserved bit,

- an exceptional event field,
- a service request enable field, and
- a reserved bit.

Figure 3-2 shows the status bits of ADB register 3.

Figure 3-2 The Status Bits of ADB Register 3



The next four bits contain the ADB device address, which you cannot change via a programming interface. The device address is a 4-bit bus address.

The second byte of register 3 contains the ADB device handler ID. The device handler ID is an 8-bit value that identifies the specific device type or its mode of operation. For example, an Apple Standard keyboard has a device handler ID of 1, while an Apple Extended keyboard has a device handler ID of 2.

ADBGetStatusBits

Obtains the status bits of device register 3.

```
OSStatus ADBGetStatusBits (
                    ADBConnectionID connection,
                    Byte *bits);
```

`connection`	An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose register 3 status bits you wish to retrieve.
`bits`	A pointer to a byte. On output, the `ADBGetStatusBits` function supplies the byte that contains the 4 most significant bits of the first byte of device register 3.
function result	An operating system status code. If you specify an incorrect connection ID, `ADBGetStatusBits` returns the `adbInvalidConnectionIDErr` status code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBGetStatusBits` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For a thorough discussion of status bits, see "Getting and Setting ADB Status Bits" (page 3-30).

To set the status bits in register 3, you can use the `ADBSetStatusBits` function (page 3-33).

To retrieve the contents of any ADB register, you use the `ADBGetRegister` function (page 3-24). To set the contents of ADB registers 0, 1, and 2, you use the `ADBSetRegister` function (page 3-26). For an overview of the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23).

To set the handler ID for ADB register 3, you can use the `ADBSetHandlerID` function (page 3-29). To obtain the handler ID for register 3, you can use the `ADBGetHandlerID` function (page 3-28). For a discussion of handler IDs, see "Getting and Setting Handler IDs" (page 3-27).

ADBSetStatusBits

Sets the status bits in ADB register 3.

```
OSStatus ADBSetStatusBits (
                    ADBConnectionID connection,
                    Byte bits);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection whose register 3 status bits you wish to set.

`bits` A byte that contains the 4 most significant bits of the first byte of device register 3.

function result An operating system status code. If you specify an incorrect connection ID, `ADBSetStatusBits` returns the `adbInvalidConnectionIDErr` status code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBSetStatusBits` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For a thorough discussion of status bits, see "Getting and Setting ADB Status Bits" (page 3-30).

To obtain the status bits in register 3, you can use the `ADBGetStatusBits` function (page 3-31).

To retrieve the contents of any ADB register, you use the `ADBGetRegister` function (page 3-24). To set the contents of ADB registers 0, 1, and 2, you use

the ADBSetRegister function (page 3-26). For an overview of the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23).

To set the handler ID for ADB register 3, you can use the ADBSetHandlerID function (page 3-29). To obtain the handler ID for register 3, you can use the ADBGetHandlerID function (page 3-28). For a discussion of handler IDs, see "Getting and Setting Handler IDs" (page 3-27)

Autopolling

Autopolling is the primary method the host computer uses to fetch data from ADB devices. Devices cannot initiate transactions; they can only respond to commands from the host. When a device has new data to transmit, it must wait until a command is issued to some ADB device. Then it should assert a service request.

The host then begins polling the addresses that have ADB devices, asking each in turn to send the contents of register 0, until the service request is no longer asserted. For details on register 0, see "Getting and Setting the ADB Registers" (page 3-23).

This section discusses the ADBGetNextAutopoll function (page 3-34), which clients call each time they are waiting for the user to perform an action.

Note
Autopolling on a specific device can be disabled. To disable the service request, you should set bit 13 of register 3 to 0. For more on register 3, see "Getting and Setting ADB Status Bits" (page 3-30) ◆

ADBGetNextAutopoll

Returns the next autopoll event.

```
OSStatus ADBGetNextAutopoll (
                    ADBConnectionID connection,
                    Duration timeOut
                    ADBRegisterContents *contents,
                    AbsoluteTime *timestamp, );
```

`connection`	An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection for which you want the ADB family to return the next autopoll. event.
`timeOut`	A duration value (expressed in milliseconds). You set this value to indicate how long you are willing to wait before an autopoll event occurs. (If you specify an invalid duration in this parameter, the `ADBGetNextAutopoll` function returns a `kernelTimeoutErr` result code in its function result.)
`contents`	A pointer to a structure of type `ADBRegisterContents` (page 3-9). On output, the `ADBGetNextAutopoll` function sets the structure to the contents of register 0 as specified in the `length` and `data` fields of the ADB Register Contents data structure. For details on register 0, see "Getting and Setting the ADB Registers" (page 3-23).
`timestamp`	A pointer to an absolute time value. On output, the `ADBGetNextAutopoll` function specifies the time at which the data arrived (taken at hardware interrupt time).
function result	An operating system status code. Your request to retrieve the next autopoll event can fail if the autopoll has already been terminated, if an invalid connection ID has been specified, or if an invalid duration has been specified. If a client called the `ADBClose` function (page 3-23) while `ADBGetNextAutopoll` is pending (for example, using another thread), the function returns the `adbConnectionTerminatedErr` status code. If you specify a connection ID incorrectly, `ADBGetNextAutopoll` returns the `adbInvalidConnectionIDErr` result code. If the duration specified in the `timeout` parameter expires before the autopoll event happens, `ADBGetNextAutopoll` returns a `kernelTimeoutErr` result code. See "ADB Result Codes" (page 3-52) for a list of the result codes the ADB family can return.

DISCUSSION

You call the `ADBGetNextAutopoll` function to wait for the user to do something, (such as press a mouse button). `ADBGetNextAutopoll` doesn't return the contents of the autopoll event until the user performs an action. Since, theoretically, it may be forever until the user does something, the `timeOut` parameter allows

you to specify how long you are willing to wait until `ADBGetNextAutopoll` retrieves the next autopoll event.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBGetNextAutopoll` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For an overview of the autopolling process, see "Autopolling" (page 3-34).

Flushing the ADB

ADBFlush

Flushes an ADB device.

```
OSStatus ADBFlush (ADBConnectionID connection);
```

`connection` An ADB connection ID. You set this value of type `ADBConnectionID` (page 3-8) to identify the ADB connection you wish to flush.

function result An operating system status code. If you specify an incorrect connection ID, `ADBFlush` returns the `adbinvalidConnectionIDErr` status code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

Those clients familiar with the System 7 Flush command may want to know that a call to `ADBFlush` is the equivalent of sending a flush command over the bus. What this Flush command does is device specific. (See the *ADB Specification* for details on the Flush command.)

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBFlush` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Resetting the ADB

If the ADB family detects a device being added to the ADB, the ADB family performs a bus reset by calling the `ADBResetBus` function (page 3-37), which destroys all existing connections. Subsequently, new connections need to be made.

ADBResetBus

Destroys all existing ADB connections.

```
OSStatus ADBResetBus (void);
```

function result An operating system status code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

When a bus reset occurs, the ADB family performs the following sequence of actions:

1. sends a device-removed notification to other parts of the I/O system for all known devices,
2. sends a reset signal over the ADB bus,
3. scans for all ADB devices and performs address resolution so that each address has at most one device, and
4. posts device added notifications for each ADB device found.

Note
`ADBResetBus` occurs automatically at start-up. Clients don't typically need to use `ADBResetBus`. ◆

Those clients familiar with the System 7 SendReset command may want to know that a call to `ADBResetBus` is the equivalent of sending a SendReset command over the bus. For details on SendReset, see the *ADB Specification*.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `ADBResetBus` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Functions Exported by ADB Family

The functions described in this function are implemented and exported by the ADB family for use only by plug-ins; each function begins with the prefix *ADBFam*.

▲ **WARNING**
The functions described in this section are only to be called by plug-ins to the ADB Family. If any other clients use these functions, the system will crash! ▲

Communication between the ADB family and its plug-ins occurs in the following sequence:

1. The ADB family calls one of plug-in functions requesting some information.
2. A plug-in starts an operation and returns.
3. Later, when an operation is complete, a plug-in calls the `ADBFamRequestComplete` function (page 3-39) to notify the family.

In some circumstances, a client does not ask for information; however, the user initiates an event. Consequently, an interrupt occurs, and the plug-in calls the `ADBFamAutopollArrived` function (page 3-40).

ADBFamRequestComplete

Indicates that an ADB family I/O operation has been completed.

```
void ADBFamRequestComplete (OSStatus theStatus);
```

`theStatus` An operating system status message that indicates the status of the request that is completing. `ADBFamRequestComplete` returns the `noErr` result code if the request has been successfully completed or the `ioErr` result code if there has been an I/O problem. If the device has timed out, the function returns the `adbDeviceTimeoutErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

The plug-in must always call the `ADBFamRequestComplete` function when I/O is complete. Moreover, the plug-in must call the `ADBFamRequestComplete` function even if the I/O operation is done immediately or in the interrupt service routine that occurs when I/O is complete. The ADB family only permits one outstanding request at a time.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

The `ADBFamRequestComplete` function is only called by ADB family plug-ins. If anyone else calls this function, the system will crash.

ADBFamAutopollArrived

Indicates that an ADB family autopoll event has arrived.

```
void ADBFamAutopollArrived (
                    Byte deviceAddress,
                    const ADBRegisterContents *contents);
```

`deviceAddress` A byte that specifies the address of the device from which the autopoll event has arrived.

`contents` A pointer to a data structure defined by type `ADBRegisterContents` (page 3-9). This structure describes the contents of ADB register 0.

DISCUSSION

A plug-in uses the `ADBFamAutopollArrived` function to inform the ADB family that an autopoll event has arrived. Examples of an autopoll events would include mouse movement information or keyboard data that arrives when the user moves the mouse.

IMPORTANT

An ADB plug-in must not call `ADBFamAutopollArrived` until it has called `MyADBPluginAutopollEnabled` (page 3-47). Furthermore, while autopolling is disabled, a plug-in must not call `ADBFamAutopollArrived`. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

The `ADBFamAutopollArrived` function is only called by ADB family plug-ins. If any other clients calls this function, the system will crash.

ADB Plug-in Defined Functions

All ADB plug-ins must implement these functions (which all begin with *MyADBPlugin*) for the ADB family to call. The ADB family provides a flexible interface, which allows plug-ins to generate, and clients to receive, device-specific data.

Validating Hardware

The ADB family can instruct a plug-in to validate an ADB device using the `MyADBPluginValidateHardwareProc` function (page 3-41). When a plug-in validates hardware, it determines if it can manage a specified Apple Desktop Bus controller.

MyADBPluginValidateHardwareProc

Instructs the plug-in to verify that the Apple Desktop Bus specified by the I/O device reference is the piece of hardware expected.

```
OSStatus MyADBPluginValidateHardwareProc
                    (IODeviceRef *device,
                    Boolean *isMyDevice);
```

`device`	A pointer to an I/O device reference, defined by the `IODeviceRef` data type, that identifies the plug-in whose associated Apple Desktop Bus is to be validated. For details on I/O device references, see "About the I/O Architecture" (page 1-3)
`isMyDevice`	A pointer to a Boolean value. On output, the plug-in sets the value to `true` if the Apple Desktop Bus identified by the I/O device reference in the `device` parameter is the one that is expected. Otherwise, the plug-in sets this parameter to `false`.
function result	An operating system status code. Your plug-in should return the result code `noErr` to indicate that its validate hardware function has been successful. If an error occurs, it should return an appropriate result code, for instance, the `paramErr` result code if an invalid I/O device reference has been specified. See "ADB Result Codes" (page 3-52) for a list of the result codes the pointing family can return.

DISCUSSION

The ADB family calls the `MyADBPluginValidateHardwareProc` function before calling the plug-in's init function.

SEE ALSO

The `ADBPluginValidateHardwareProc` type (page 3-13) defines an ADB family plug-in's validate hardware function.

Initializing ADB Plug-ins

MyADBPluginInitProc

Instructs the plug-in code to initialize both its software structures and its Apple Desktop Bus controller.

```
OSStatus MyADBPluginInitProc (IODeviceRef *device);
```

`device`	A pointer to the I/O device reference that identifies the Apple Desktop Bus controller to be initialized. For more on I/O device references, see "About the I/O Architecture" (page 1-3)
function result	An operating system status code. Your plug-in should return the result code `noErr` to indicate that the init function has been successful. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

The ADB family calls `MyADBPluginValidateHardwareProc` (page 3-41) before it makes any other calls to the plug-in. At initialization, the plug-in should initialize any autopolling delay to approximately 1/100 second as well as disable autopolling and empty the autopoll list.

SEE ALSO

The `ADBPluginInitProc` type (page 3-14) defines the `MyADBPluginInitProc` function.

Setting and Getting Autopoll Delay

The **autopoll delay** is the interval between autopoll commands from the Apple Desktop Bus controller. When the ADB family sets autopolling delay using the `MyADBPluginSetAutopollDelayProc` function (page 3-43), the plug-in uses the closest value supported by the hardware, never greater than 16.625 milliseconds. The ADB family calls the `MyADBPluginGetAutopollDelayProc` function (page 3-44) to obtain the actual value.

MyADBPluginSetAutopollDelayProc

Instructs the plug-in code to set the autopoll delay.

```
OSStatus MyADBPluginSetAutopollDelayProc (Duration delay);
```

`delay` A duration value that specifies the autopoll interval, that is, the closest value for delay supported by the Apple Desktop Bus controller. To obtain this value, call the function `MyADBPluginGetAutopollDelayProc` (page 3-44).

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that the set autopoll delay function has been successful. If an error occurs, it should return an appropriate result code, for instance, the `paramErr` result code if the ADB family has specified an invalid duration. If an I/O error has occurred, the function should return `ioErr`. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginSetAutopollDelayProc` type (page 3-15) defines the ADB plug-in set autopoll delay function.

To instruct a plug-in to obtain an autopoll delay, the ADB family can use the `MyADBPluginGetAutopollDelayProc` function (page 3-44). For details on autopoll delay, see "Setting and Getting Autopoll Delay" (page 3-43).

MyADBPluginGetAutopollDelayProc

Instructs the plug-in to obtain an actual autopoll delay.

```
OSStatus MyADBPluginGetAutopollDelayProc (Duration *delay);
```

`delay` A pointer to a duration value. On output, the plug-in sets value to the closest value for autopoll delay supported by the Apple Desktop Bus controller.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its get autopoll delay function has been successful. If an error occurs, it should return an appropriate result code, for instance, the `paramErr` result code if the ADB family has specified an invalid duration. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginGetAutopollDelayProc` type (page 3-15) defines the ADB plug-in get autopoll delay function.

To instruct a plug-in to set an autopoll delay, the ADB family can use the `MyADBPluginSetAutopollDelayProc` function (page 3-43). For details on autopoll delay, see "Setting and Getting Autopoll Delay" (page 3-43).

Setting and Getting the Autopoll List

The autopoll list consists of group of addresses polled during autopoll operations. Typically, the autopoll list consists of all the devices that have been opened using the `ADBOpen` function (page 3-21). The ADB family can use the `MyADBPluginSetAutopollList` (page 3-45) and `MyADBPluginGetAutopollList` (page 3-46) functions to instruct a plug-in to set and obtain the autopoll list respectively.

MyADBPluginSetAutopollList

Instructs the plug-in to set the autopoll list.

```
OSStatus MyADBPluginSetAutopollList (UInt16 addressMask);
```

`addressMask` An unsigned 16-bit integer that represents an address mask. Each bit of the address mask describes an ADB address with the least significant bit at address 0 and the most significant bit at address 15. There is one bit per address and 16 possible addresses.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its set autopoll list function has been successful. If an I/O error has occurred, the function should return the `ioErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginSetAutopollListProc` type (page 3-15) defines the ADB plug-in set autopoll list function.

To instruct a plug-in to obtain an autopoll list, the ADB family can use the `MyADBPluginGetAutopollListProc` function (page 3-46).

MyADBPluginGetAutopollListProc

Instructs the plug-in to obtain the autopoll list.

```
OSStatus MyADBPluginGetAutopollListProc (UInt16 *addressMask);
```

`addressMask` A pointer to an unsigned 16-bit integer that represents an address mask. On output, the plug-in sets each bit of the parameter to describe an ADB address with the least significant bit at address 0 and the most significant bit at address 15. There is one bit per address and 16 possible addresses.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its get autopoll list function has been successful. If an error occurs, it should return an appropriate result code, for instance, `paramErr`. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginGetAutopollListProc` type (page 3-16) defines the ADB plug-in get autopoll list function.

To instruct a plug-in to set an autopoll list, the ADB family can use the `MyADBPluginSetAutopollListProc` function (page 3-45).

Enabling and Disabling Autopolling

The ADB family calls the `MyADBPluginAutopollEnableProc` (page 3-47) and `MyADBPluginAutopollDisableProc` (page 3-47) functions to instruct the plug-in to turn autopolling on and off respectively.

MyADBPluginAutopollEnableProc

Instructs the plug-in to turn on autopolling

```
OSStatus MyADBPluginAutopollEnableProc (void);
```

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its autopoll enable function has been successful. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginAutopollEnableProc` type (page 3-16) defines the ADB plug-in enable autopolling function.

To instruct a plug-in to turnoff autopolling, the ADB family can use the `MyADBPluginAutopollDisableProc` function (page 3-47).

MyADBPluginAutopollDisableProc

Instructs the plug-in to turn off autopolling.

```
OSStatus MyADBPluginAutopollDisableProc (void);
```

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its autopoll disable function has been successfully activated. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

To instruct a plug-in to turn on autopolling, the ADB family can use the `MyADBPluginAutopollEnable` function (page 3-47).

The `ADBPluginAutopollDisableProc` type (page 3-16) defines the ADB plug-in disable autopolling function.

Resetting the ADB Bus

MyADBPluginResetBusProc

Instructs the plug-in to reset the ADB bus.

```
OSStatus MyADBPluginResetBus (void);
```

function result An operating system status code. An operating system status code. Your plug-in should return the result code `noErr` to indicate that its flush function has been successful. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginResetBusProc` type (page 3-17) defines the ADB plug-in reset bus function.

For a description of resetting the ADB, see "Resetting the ADB" (page 3-37).

Flushing ADB Devices

MyADBPluginFlushProc

Instructs a plug-in to flush an ADB device.

```
OSStatus MyADBPluginFlushProc (Byte deviceAddress);
```

`deviceAddress` A byte that indicates the address of the device the ADB family wants to flush.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its flush function has been successful. If an error occurs, it should return an appropriate

result code, for instance, the `paramErr` result code if the ADB family has specified an invalid device address. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginFlushProc` type (page 3-17) defines the ADB plug-in flush function.

Setting and Getting the ADB Plug-in Register

The ADB family calls the `MyADBPluginSetRegisterProc` and `MyADBPluginGetRegisterProc` functions to instruct a plug-in to set and get the contents of a register on an ADB device. For details on the ADB registers, see "Getting and Setting the ADB Registers" (page 3-23).

MyADBPluginSetRegisterProc

Instructs the plug-in to set the contents of any of the ADB registers.

```
OSStatus MyADBPluginSetRegisterProc (
                    Byte deviceAddress,
                    Byte registerNumber,
                    const ADBRegisterContents *contents);
```

`deviceAddress` A byte that indicates the address of the device whose register the ADB family wants to set.

`registerNumber`
A byte that describes the register whose contents the ADB family wants to set.

`contents` A pointer to a structure of type `ADBRegisterContents` (page 3-9). The structure describes the contents of the data to be set in the ADB register specified in the `registerNumber` parameter.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its set register function has been successful. If an error occurs, it should return an

appropriate result code, for instance, the `paramErr` result code if the ADB family has specified an invalid device address or register number. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginSetRegisterProc` type (page 3-18) defines the ADB plug-in set register function.

To instruct a plug-in to obtain the contents of a specified register on a particular ADB device, the ADB family calls `MyADBPluginGetRegisterProc` (page 3-50).

MyADBPluginGetRegisterProc

Instructs the plug-in to retrieve the contents of an ADB register for a specified device.

```
OSStatus MyADBPluginGetRegisterProc (
                    Byte deviceAddress,
                    Byte registerNumber,
                    ADBRegisterContents *contents);
```

`deviceAddress` A byte that indicates the address of the device whose register the ADB family wants to get.

`registerNumber` A byte that describes the register whose contents the ADB family wants to get.

`contents` A pointer to a structure of type `ADBRegisterContents` (page 3-9). On output, the plug-in provides in the structure the contents of the ADB register specified in the `registerNumber` parameter.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its get register function has been successful. If an error occurs, it should return an appropriate result code, for instance, the `paramErr` result code if the ADB family has specified an invalid device address or register number. If there is no response from the specified

device address, your plug-in should return the `adbDeviceTimeoutErr` result code. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

SEE ALSO

The `ADBPluginGetRegisterProc` type (page 3-18) defines the ADB plug-in get register function.

To instruct a plug-in to set the contents of a specified register on a particular ADB device, the ADB family calls `MyADBPluginSetRegisterProc` (page 3-49).

Setting the Keyboard List

The `MyADBPluginSetKeyboardList` function instructs the plug-in to tell the Apple Desktop Bus controller which device addresses are keyboards. For more on keyboard lists, see "ADBPluginSetAutopollListProc" (page 3-15).

MyADBPluginSetKeyboardList

Instructs the plug-in to set a keyboard list.

```
OSStatus MyADBPluginSetKeyboardList (UInt16 addressMask);
```

`addressMask` An unsigned 16-bit integer that represents an address mask. Each bit of the parameter describes an ADB address with the least significant bit at address 0, the most significant bit at address 15. The mask describes 1 bit per address and 16 possible addresses.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its set keyboard list function has been successful. If an error occurs, it should return an appropriate result code, for instance, the `paramErr` result code if the ADB family has specified an invalid address mask. See "ADB Result Codes" (page 3-52) for a list of possible result codes.

DISCUSSION

When the ADB family wants to create a list of the ADB addresses that are keyboards, it calls `MyADBPluginSetKeybordListProc`.

SEE ALSO

The `ADBPluginSetKeyboardListProc` type (page 3-18) defines the ADB plug-in set keyboard list function.

See "Setting the Keyboard List" (page 3-51) for a definition of the ADB keyboard list and details on its use.

ADB Result Codes

Many ADB family functions return result codes. The various result codes specific to the ADB family are listed here.

`adbDeviceBusyErr`	-30279	ADB device has already been opened.
`adbInvalidConnectionIDErr`	-30278	ADB connection ID has been incorrectly specified.
`adbConnectionTerminatedErr`	-30277	ADB device has been closed or unplugged during the call.
`adbDeviceTimeoutErr`	-30276	ADB device has timed out.
`adbReservedHandlerIDErr`	-30275	Specified ADB handler ID has already been reserved.
`adbInvalidHandlerIDErr`	-30274	ADB handler ID has been incorrectly specified.

Glossary

ADB clients Software receiving services from the ADB family (for instance, the keyboard and pointing families as well as applications running in user space)

ADB connection A logical path to an ADB device and serves to control access to the device. Clients can obtain an ID for an ADB connection by calling the `ADBOpen` function (page 3-21).

ADB control register The name for ADB register 3. Two bytes in length, the control register stores the device address, handler ID, and four status bits. The *ADB Specification* describes the protocol for changing these fields. See also **ADB device address**, **ADB handler ID**, and **ADB status bits**.

ADB device Any peripheral that can connect to the ADB and meets the design requirements described in the "Apple Desktop Bus Controller" chapter of *Macintosh Technology in the Common Hardware Reference Platform*, published by Morgan Kaufman.

ADB device address The second two bits of the ADB control register (ADB register 3). You cannot change the ADB device address via a programming interface. This 4-bit bus address uniquely identifies devices of the same type. See also **ADB control register**.

ADB device registers Up to four registers for storing data provided to each device connected to the Apple Desktop Bus. An ADB device can implement these registers as it chooses; that is, an ADB register does not have to correspond to an actual hardware register on the ADB device. An ADB device is accessed over the ADB by reading from or writing to these registers. Each ADB device register may store between 2 and 8 bytes of data. The ADB family defines the `ADBRegisterContents` data type (page 3-9) to provide information about the contents of an ADB register.

ADB family Software that allows clients to get information about and communicate with hardware devices attached to the Apple Desktop Bus (ADB).

ADB handler ID An ADB device-specific 8-bit value in the control register. Taken together, a device's default address and handler ID uniquely identify the particular data protocol the device uses for communication. Devices may support more than one protocol. The handler ID for a device occupies the second byte of the control register (ADB register 3). You can use the `ADBGetHandlerID` (page 3-28) and `ADBSetHandlerID` (page 3-29) functions, respectively, to obtain and set the handler IDs. See also **ADB device address** and **ADB control register**.

ADB keyboard list A list of the ADB addresses that have keyboards. When the ADB family wants to set a keyboard list, it calls the set keyboard list function (page 3-51) provided by the plug-in.

ADB plug-ins Software modules, such as drivers for specific families of computers, such as the 6100, 7100, 8100; the 7500, 8500, 9500; or the Powerbook 5300. Whereas the ADB family provides services to clients, the ADB plug-ins actually implement the request for services

ADB status bits The 4 most significant bits of the control register (also called ADB register 3). The upper four bits of ADB register 3 contain the status bits, which consist of a service request enable field, an exceptional event field, and several reserved bits. The `ADBGetStatusBits` (page 3-31) and `ADBSetStatusBits` (page 3-33) functions get and set the status bits. For an illustration of the ADB status bits, see Figure 3-2 (page 3-31).

Apple Desktop Bus (**ADB**) An open-collector, low-speed serial bus that connects user input peripherals such as keyboards, mice, graphics tablets, and joysticks to a host computer or to other hardware equipment. Macintosh computers come equipped with one or two ADB connectors. Although a particular model might include two ADB connectors, all models come with only one Apple Desktop Bus. The ADB is Apple Computer's standard interface for input devices such as keyboards and mouse devices.

Apple Desktop Bus controller Usually a microcontroller in the host computer that actually performs the communication of the bus to the ADB devices.

autopolling The primary method the ADB hardware uses to fetch data from ADB devices. Devices cannot initiate transactions; they can only respond to commands from Apple Desktop Bus controller.

autopoll delay The interval between autopoll commands performed by the Apple Desktop Bus controller. When the ADB family sets autopolling delay using the `MyADBPluginSetAutopollDelayProc` function (page 3-43), the plug-in uses the closest value supported by the Apple Desktop Bus controller, never greater than 16.625 milliseconds. The ADB family calls the `MyADBPluginGetAutopollDelayProc` function (page 3-44) to return the actual value.

autopoll list A group of addresses that are polled during autopoll operations. Typically, the autopoll list consists of all the devices that have been opened using the `ADBOpen` function (page 3-21). When the ADB family wants to set the autopoll list, it calls the set autopoll list function (page 3-45) provided by the plug-in.

Listen command A command from the Apple Desktop Bus controller that instructs a device to prepare to receive additional data. When a client calls the `ADBSetRegister` function (page 3-26), it is the equivalent of the Apple Desktop Bus controller's issuing a Listen command to the device.

Talk command A command from the Apple Desktop Bus controller that fetches user input or other data from an ADB device. A talk command requests

that the specified device send the contents of a specified device register across the bus. When clients call the `ADBGetRegister` function (page 3-24), it is the equivalent of the Apple Desktop Bus controller's issuing a talk command to the device.

Pointing Family Reference

Contents

This chapter describes the **pointing family**, which provides support for pointing devices in Mac OS 8. **Pointing devices** are user-input peripherals (such as mice, tablets, and joysticks) that indicate position and orientation and facilitate movement through the user interface. Pointing devices are commonly used to control cursors and to control objects in space.

Note
In subsequent developer releases of Mac OS 8, the keyboard and pointing families will probably become part of an input devices family. ◆

Most applications use the Apple Event Manager to obtain rudimentary information about the mouse and the system cursor, which is adequate pointing data for most clients. For complete information about receiving and interpreting mouse input, see *Apple Events in Mac OS 8*. Any applications that require more complicated data from pointing devices other than the mouse and the system cursor can use the functions described in "Pointing Family Client Functions" (page 4-27).

Pointing family clients include applications, such as graphics and paint programs and games that often take advantage of the special capabilities of pointing devices. In addition, system software, for instance, the Apple Event Manager and graphics systems such as QuickDraw are clients of the pointing family. Moreover, applications that provide control panels sometimes specify the behavior of pointing devices. If you are a pointing family client, you will find most of the information of interest in"Pointing Family Client Functions" (page 4-27).

Pointing family plug-ins are drivers that control the pointing devices themselves (such as mice, tablets, joysticks, and 3D trackballs). If you are implementing a pointing device driver, most of the information of interest is in "Pointing Family Plug-In-Defined Functions" (page 4-54).

About the Pointing Family

The pointing family supports pointing devices by

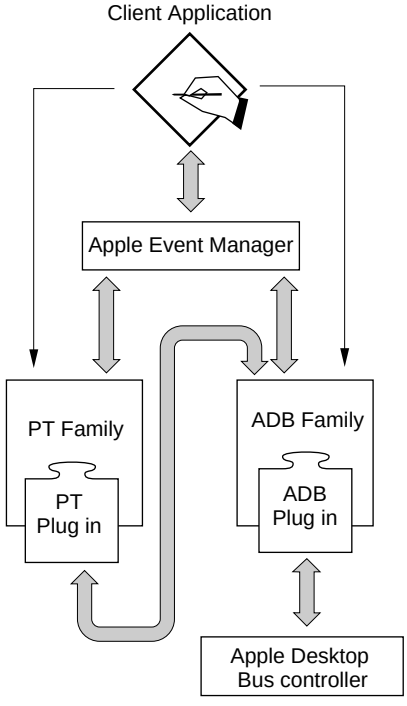
- distributing data from pointing devices to system software and application clients

- providing clients with a common interface to all pointing devices.
- applying standard manipulations of pointing data before providing data to clients
- keeping track of state or buffered data, according to client preferences
- allowing users to attach multiple pointing devices to their computers
- permitting applications to register for control of pointing devices and set device modes

Figure 4-1 illustrates how a client can use the pointing family and its plug-ins to get information about pointing devices.

- If you are a pointing family client application, you can call the pointing family programming interface described in "Pointing Family Client Functions" (page 4-27). (All the pointing family client functions begin with the prefix *PT*.) In certain situations, clients can call the ADB family programming interface. (For details on the ADB family interface, see "ADB Family Reference" (page 3-5).)
- If you are designing a new pointing device, you need to write a pointing family plug-in that communicates with the device and to implement the pointing family plug-in programming interface described in "Pointing Family Plug-In-Defined Functions" (page 4-54). (All the pointing family plug-in defined functions begin with the prefix *MyPTPlugin*.) The pointing family calls these functions. If your device is an ADB peripheral, your pointing family plug-in can call the ADB family programming interface, which, in turn, communicates with the ADB family and the Apple Desktop Bus controller. For more information on the Apple Desktop Bus controller, see "ADB Family Reference" (page 3-5).

Figure 4-1 The Pointing Family, Its Clients, and Plug-ins



Note
In System 7, whatever pointing device you use affects the system cursor. For this developer release of Mac OS 8 only, any device that is a mouse device class drives the system cursor. See "Pointing Device Class" (page 4-15) for details on pointing device classes. ◆

If you want to test another pointing device besides the mouse, you must write a plug-in for the device and then write an application client that registers information for that device using the `PTRegisterNewTracker` function (page 4-33).

If you want to test another pointing device that registers as a mouse and you want to obtain data, you must obtain that information through the Apple Event Manager. (For more on the Apple Event Manager, see *Apple Events in Mac OS 8*.)

Note
In subsequent developer releases of Mac OS 8, applications can determine whether or not a device controls the system cursor all the time or some of the time. ◆

Constants and Data Types

This section describes the data types and constants used in the pointing family programming interface and in the plug-in interface. A client, generally an application or system software, uses the services of the pointing family and its plug-ins to manage data generated by pointing devices.

Pointing Family Tracker Reference

Pointing family trackers represent connections between pointing devices and clients. Each tracker stores information (for instance, the state and buffer of the tracker currently associated with the plug-in's controller) about what kind of data the client wants and how the client would like to receive it.The pointing family defines a unique reference for trackers. Clients can use the tracker

reference to specify a device in many pointing family functions. A tracker reference is specified by the `PTTrackerRef` data type.

```
typedef struct OpaquePTTrackerRef* PTTrackerRef;
```

Pointing Data Structure

Pointing devices generate widely divergent kinds of information. Most devices provide the two-dimensional mouse location and the up and down notification of one button. Other classes of devices share other sorts of data. For instance, tablets generate pressure and tilt, joysticks generate data to emulate controls like throttles and brakes, and 3D-trackballs generate z-axis data. In addition, particular devices may generate unique device-specific information. To support this set of highly variable data, the pointing family provides the pointing data structure (defined by the `PTData` type).

Each pointing data structure represents a piece of data from a pointing device. The pointing family carries this information from each plug-in through the family to the clients.

The pointing data structure is a variable-length structure with a `uniqueDeviceData` field, which lets the pointing family pass device-specific information to clients that request it.

Note
Future developer releases of Mac OS 8 will probably contain better support for an application client's understanding of device data. Future versions of the operating system will most likely define different kinds of data that different devices generate, so applications can understand, for example, generic tablet data, rather than only having access to a specific manufacturer's description of that data. ◆

```
struct PTData {
    AbsoluteTime    sequencingTag;      /* sequencing tag */
    PTPosition      position;           /* position */
    PTButtonState   buttons;            /* buttons */
    void *          uniqueDeviceData;   /* variable-length unique
                                           device data */
};
```

```
typedef struct PTData PTData;

typedef PTData *PTDataPtr;
```

Field descriptions

`sequencingTag` An absolute time value that specifies the time that the data was generated. This value is used for sequencing data.

`position` The pointing position data structure defined by the `PTPosition` data type (page 4-10). This structure identifies the coordinates of the position of the device.

`buttons` The 32 bits of a `PTButtonState` type (page 4-11) represent the state of 32 buttons. Bits are set to indicate that the corresponding button is pressed.

Note
For this developer release of Mac OS 8, if any of the bits in a `PTButtonState` type are set, it is considered to be a mouse-up. Otherwise, it is considered to be a mouse-down. ◆

`uniqueDeviceData` Variable-length, device-specific information for the pointing family to pass to clients that request it.

Pointing Position Structure

The pointing position structure is defined by the `PTPosition` data type. The pointing position structure represents the three-dimensional position of a pointing device. Clients can use the pointing position data type in the `position` field of the pointing data structure (page 4-9).

```
struct PTPosition {
    signed long         x;  /* x-coordinate */
    signed long         y;  /* y-coordinate */
    signed long         z;  /* z-coordinate */
};

typedef struct PTPosition PTPosition;

typedef PTPosition *PTPositionPtr;
```

Field descriptions

x	The value specifying the x-coordinate of a pointing device.
y	The value specifying the y-coordinate of a pointing device.
z	The value specifying the z-coordinate of a pointing device.

Pointing Button State Type

The pointing button state, defined by the `PTButtonState` data type, an unsigned long integer, represents the state of 32 buttons. Clients can use the pointing button state structure in the `buttons` field of the pointing data structure (page 4-9) to indicate which button is pressed. The Apple Event Manager assumes the first bit is the mouse button. Most devices do not require 32 buttons.

```
typedef unsigned long PTButtonState;

typedef PTButtonState *PTButtonStatePtr;
```

Pointing Device Modes Structure

Pointing devices differ in the kinds of mode information they have**. Device mode data** is the kind of information that pointing family clients pass to a pointing device, usually to set a mode on the device. Examples of device modes include tactile feedback for steering wheels, absolute versus relative mode for tablets, and LED displays. The pointing family supports device mode data with the pointing device modes structure, which is defined by the `PTDeviceModes` data type.

The plug-in maintains its own modes based on information it may obtain from clients and data it may get from its device. At start-up, plug-ins notify the pointing family of the size of the device mode data in the pointing device capabilities structure, defined by the `PTDeviceCapabilities` data type (page 4-14).

A pointing device modes structure contains all the mode information for a device. Clients typically set device modes. This structure is of variable length.

```
struct PTDeviceModes {
    PTDataRelation      relation;           /* data relation constant */
    void *              uniqueDeviceModes;  /* unique device modes */
};

typedef struct PTDeviceModes PTDeviceModes;

typedef PTDeviceModes *PTDeviceModesPtr;
```

Field descriptions

`relation` A data relation enumerator defined by the `PTDataRelation` data type (page 4-12). These enumerators indicate if the data clients pass back to the device can be absolute, relative, or either absolute or relative.

`uniqueDeviceModes`

A variable-length field of device-specific mode data. Plug-ins need to let applications know to leave enough space here for device data. Device-specific modes should be stored following the pointing device modes structure. Plug-ins with more modes should define structures of their own that contain `PTDeviceModes` data types (page 4-11).

Data Relation Enumerators

At initialization, a pointing device indicates in its pointing device capabilities structure, defined by the `PTDeviceCapabilities` data type (page 4-14), whether it generates absolute data, relative data, or either kind of information. It also indicates whether its default data is absolute or relative. A device sets up its modes and puts the default kind of data in its device modes structure, defined by the `PTDeviceModes` data type (page 4-11).

To find out what kind of data a device can generate, clients can use the `PTGetDeviceCapabilities` function (page 4-29). To determine what kind of data a device is currently generating, clients can use the `PTGetDeviceModes` function (page 4-36). If the device can generate either absolute or relative data, clients can change the kind of data currently being generated by using the `PTSetDeviceModes` function (page 4-35).

The data relation enumerators define whether a device generates absolute data, relative data, or either kind of data. This information is especially relevant for tablets.

Absolute data is position information sent to the pointing family by a plug-in that generates the actual coordinates of a pointing device, for instance, (10, 20). **Relative data** is position information sent to the pointing family by a plug-in that describes how far the pointing device moved from an already established coordinate. For instance, if the coordinate was (10, 20), it might have changed to (11, 21). In this case, the data (1,1) would be generated.

Note
The pointing family handles position information differently for absolute and relative data. When it updates the state of absolute data, it replaces what was there before. When it updates the state of relative data, it adds that information to the pre-existing coordinates. ◆

The data relation enumerators are defined in the `PTDataRelation` type, an unsigned 16-bit integer. Clients use the data relation enumerators in the `relation` field of the pointing device mode structure (page 4-11) to define what kind of data they want, and devices use them to define what they are capable of generating in the device capabilities structure (page 4-14). Some devices can generate either, depending on what the client wants.

```
typedef UInt16 PTDataRelation;

enum {
    kAbsoluteData           = 1,    /* absolute data */
    kRelativeData           = 2,    /* relative data */
    kAbsoluteOrRelativeData = 3     /* absolute or relative data */
};
```

Enumerator descriptions

`kAbsoluteData` The absolute data enumerator. Use this enumerator in the `relation` field of the pointing device modes structure (page 4-11) to indicate that the kind of information that the device generates is absolute.

`kRelativeData` The relative data enumerator. Use this enumerator in the `relation` field of the pointing device modes structure (page 4-11) to indicate that kind of information the device generates is relative.

`kAbsoluteOrRelativeData`
The absolute or relative enumerator. Use this enumerator in the `relation` field of the pointing device modes structure

(page 4-11) to indicate that kind of information that the device generates can be either absolute or relative.

Pointing Device Capabilities

A pointing device capabilities structure, defined by the `PTDeviceCapabilities` data type, describes a particular pointing device. At initialization, a plug-in fills in a pointing device capabilities structure.

Clients can check the pointing device capabilities structure in the `capabilities` parameter of the `PTGetDeviceCapabilities` function (page 4-29) to find out specifics about the capabilities of a specified device.

The plug-in provides a pointer to its `MyPTPluginInitialize` function (page 4-56). When the pointing family calls the plug-in's `MyPTPluginInitializePtr` function (page 4-56), the family provides a pointer to the pointing device capabilities structure. Then the plug-in fills in the fields of the structure.

```
struct PTDeviceCapabilities {
    PTDeviceClass   deviceClass;              /* device class */
    ByteCount       dataSize;                 /* data size */
    ByteCount       modeDataSize;             /* device mode data size */
    PTDataRelation  availableDataRelations;   /* available data relations */
    PTDataRelation  defaultDataRelation;      /* default data relations */
    long            latency;                  /* latency */
    Boolean         imitatesMouse;            /* Does device imitate a mouse? */
};

typedef struct PTDeviceCapabilities PTDeviceCapabilities;

typedef PTDeviceCapabilities *PTDeviceCapabilitiesPtr;
```

Field descriptions

`deviceClass` The device class of a particular pointing device. This value of type `PTDeviceClass` (page 4-15) describes the class of a particular device, for instance, a mouse, tablet, joystick, trackball, trackpad, or 3D trackball.

`dataSize`	A byte count that specifies the size of the data generated by the particular pointing device and corresponds to the variable-size pointing data structure (page 4-9).
`modeDataSize`	A byte count that specifies the size of the device's device mode data. For more on device mode data, see "Pointing Device Modes Structure" (page 4-11).
`availableDataRelations`	A pointing device data relations constant from the `PTDataRelation` enumeration (page 4-12) that specifies whether the available data is absolute, relative, or can be either absolute or relative.
`defaultDataRelation`	A pointing device data relations constant from the `PTDataRelation` enumeration (page 4-12) that specifies whether the default data is absolute, relative, or can be either absolute or relative.
`latency`	A long integer that specifies the **latency** of a particular device (that is, the time it takes a plug-in to gather up data generated by the device and send it to the pointing family) in microseconds.
`imitatesMouse`	A Boolean value that specifies whether the pointing device imitates a mouse. Set this value to `true` if the device imitates a mouse; otherwise, set this value to `false`.

Pointing Device Class

The pointing device class defined by the `PTDeviceClass` type indicates the category of a device, such as mouse, tablet, joystick, trackball, trackpad, or 3D trackball. The pointing family defines the device classes, which are OS types. Plug-ins indicate their pointing device class at initialization when they fill in the `deviceClass` field of the pointing device capabilities structure, defined by the `PTDeviceCapabilities` data type (page 4-14). Clients can check a device's class by using the `PTGetDeviceCapabilities` function (page 4-29) to obtain the device capabilities structure. They also use the `PTDeviceClass` types (page 4-15) to filter through a list of pointing devices in the `filter` parameter of the `PTGetNextDevice` function (page 4-28).

```
typedef OSType PTDeviceClass;
```

```
enum {
    kAnyDeviceClass         =   'anyp'      /* any device class */
    kMouseDeviceClass       =   'mous',     /* mouse device class */
    kTabletDeviceClass      =   'tblt',     /* tablet device class */
    kJoystickDeviceClass    =   'joys',     /* joystick device class */
    kTrackballDeviceClass   =   'trkb',,    /* trackball device class */
    kTrackpadDeviceClass    =   'trkp',     /* trackball device class */
    k3DTrackballDeviceClass =   '3dtb'      /* 3D trackball device class */
```

Enumerator descriptions

`kAnyDeviceClass` The next device class enumerator. If you don't want to specify what kind of device you want, that is, you want to iterate over all devices, use this enumerator in the `filter` parameter of the `PTGetNextDevice` function (page 4-28).

`kMouseDeviceClass` The mouse device class enumerator. Use this enumerator to indicate a mouse in the `deviceClass` field of the pointing device capabilities structure defined by the `PTDeviceCapabilities` data type (page 4-14) or in the `filter` parameter of the `PTGetNextDevice` function (page 4-28).

`kTabletDeviceClass`

The tablet device class enumerator. Use this enumerator to indicate a tablet in the `deviceClass` field of the pointing device capabilities structure defined by the `PTDeviceCapabilities` data type (page 4-14) or in the `filter` parameter of the `PTGetNextDevice` function (page 4-28).

`kJoystickDeviceClass`

The joystick device class enumerator. Use this enumerator to indicate a joystick in the `deviceClass` field of the pointing device capabilities structure defined by the `PTDeviceCapabilities` data type (page 4-14) or in the `filter` parameter of the `PTGetNextDevice` function (page 4-28).

`kTrackballDeviceClass`

The trackball device class enumerator. Use this enumerator to indicate a trackball in the `deviceClass` field of the pointing device capabilities structure defined by the `PTDeviceCapabilities` data type (page 4-14) or in the

filter parameter of the PTGetNextDevice function (page 4-28).

kTrackpadDeviceClass
: The trackpad device class enumerator. Use this enumerator to indicate a trackpad in the deviceClass field of the pointing device capabilities structure defined by the PTDeviceCapabilities data type (page 4-14) or in the filter parameter of the PTGetDeviceCapabilities function (page 4-28).

k3DTrackballDeviceClass
: The 3D-trackball device class enumerator. Use this enumerator to indicate a 3D-trackball in the deviceClass field of the pointing device capabilities structure defined by the PTDeviceCapabilities data type (page 4-14) or in the filter parameter of the PTGetNextDevice (page 4-28) or the PTGetDeviceCapabilities function (page 4-28).

Minimum Pointing Device Data Size

Plug-ins can use the minimum pointing device data size constant kMinPTDataSize as in the dataSize field of their pointing device capabilities structure (page 4-14) if they only generate the basic mouse data and position buttons. For example, a mouse usually generates 24 bytes of data, that is, the size of its sequencing tag, its position, and its buttons. On the other hand, a tablet might generate pressure and tilt in addition to a sequencing tag, position, and button data, so a plug-in would need to build an internal data structure to reflect the size of such data, then pass that size in the dataSize field of its pointing device capabilities structure.

```
enum {
    kMinPTDataSize = 24     /* minimum size of data generated by
                               a plug-in */
};
```

Enumerator description

Field descriptions

kMinPTDataSize
: The minimum pointing device data size constant. Plug-ins specify this constant in the dataSize field of the pointing

device capabilities structure to specify a value of 24 bytes. Otherwise, they need to provide their own data structures.

Pointing Device Identifier

The pointing device identifier, defined by the `PTDeviceIdentifier` type, uniquely identifies the brand and model of a device. The identifier is a pointer to a `NULL`-terminated C string defined by the manufacturer in a way that makes sense for the product line. To obtain the device identifier for a pointing device, you can use the `PTGetDeviceIdentification` function (page 4-31). The device identifier is returned in the function's `identification` parameter.

```
struct PTDeviceIdentifier {
    char    identifier[255];    /* pointing device identifier */

typedef struct PTDeviceIdentifier PTDeviceIdentifier;

typedef PTDeviceIdentifier *PTDeviceIdentifierPtr;
```

Field description

`identifier` A fixed-length string that uniquely identifies a pointing device (typically, its brand and model).

Pointing Pinning Rectangle List

Clients can specify and retrieve **pinning rectangles** for pointing family trackers. Pinning rectangles restrict a position to within their boundaries, no matter how far a device may move. A tracker's static position does not extend beyond a pinning rectangle. If a tracker has more than 1 pinning rectangle, the position can be inside any of them, but not outside all of them. When a pointing device sends data that is outside the tracker's pinning rectangles in 1 dimension, that dimension does not change. For example, the tracker that represents the system cursor has pinning rectangles that correspond to the screens. In this way, the cursor stops at the edge of the screen even if the user keeps moving the mouse in 1 direction for a long time.

Note
The pointing family only pins data from relative devices. See "Data Relation Enumerators" (page 4-12) for more on relative and absolute devices. ◆

The pointing pinning rectangle list is defined by the `PTPinningRectList` data type. Clients use this structure in the `rectList` parameter of the `PTSetPinningRects` (page 4-39) function.

```
struct PTPinningRectList {
    short           numRects;       /* number of rectangles in list */
    Rect *          pinningRect;    /* pointer to a list of rectangles */
                                    /* rectangles must be in global
                                        coordinates, in pixels */
};

typedef struct PTPinningRectList PTPinningRectList;

typedef PTPinningRectList *PTPinningRectListPtr;
```

Field descriptions

`numRects`	A short integer that specifies the number of rectangles in the pointing pinning rectangle list structure.
`pinningRect`	A pointer to an array of rectangles. The rectangles must be described in global coordinates and in pixels.

Note
In this developer release of Mac OS 8, you may only specify one pinning rectangle. ◆

Pointing Family Plug-In Data Types

This section describes the data types in the pointing family plug-in programming interface.

Pointing Family Device Dispatch Table

Each pointing family plug-in must export a pointing family device dispatch table, so the pointing family can find the functions it contains. The pointing family calls the Driver and Family Matching Software (DFM) to load each plug-in. Subsequently, the DFM returns a pointer to the dispatch table. For more on the DFM, see "Driver and Family Matching" (page 2-3).

The pointing family device dispatch table is defined by the `PTDeviceDispatchTable` data type.

```
struct PTDeviceDispatchTable {
    PTPluginHeader              header;                     /* pointing family plug-in
                                                               header */
    PTPluginValidateHardwarePtr PTPluginValidateHardware    /* validate hardware
                                                               function */
    PTPluginInitializePtr       PTPluginInitialize;         /* initialization
                                                               function */
    PTPluginTerminatePtr        PTPluginTerminate;          /* terminate function */
    PTPluginStartIOPtr          PTPluginStartIO;            /* start I/O function */
    PTPluginStopIOPtr           PTPluginStopIO;             /* stop I/O function */
    PTPluginGetNextDataPtr      PTPluginGetNextData;        /* get next data
                                                               function */
    PTPluginGetDeviceModesPtr   PTPluginGetDeviceModes      /* get device modes
                                                               function */
    PTPluginSetDeviceModesPtr   PTPluginSetDeviceModes      /* set device modes
                                                               function */
};

typedef struct PTDeviceDispatchTable PTDeviceDispatchTable;

typedef PtrDeviceDispatchTable *PTDeviceDispatchTablePtr;
```

Field descriptions

`header` The pointing family plug-in header, defined by the `PTPluginHeader` data type (page 4-21).

`PTPluginValidateHardware` A pointer to the plug-in defined validate hardware function.

`PTPluginInitialize`
A pointer to the plug-in defined initialize function (page 4-56).

`PTPluginTerminate`
A pointer to the plug-in defined terminate function (page 4-58).

`PTPluginStartIO` A pointer to the plug-in defined start I/O function (page 4-59).

`PTPluginStopIO` A pointer to the plug-in defined stop I/O function (page 4-60).

`PTPluginGetNextData`
A pointer to the plug-in defined get next data function (page 4-61).

`PTPluginGetDeviceModes`
A pointer to the plug-in defined get device modes function (page 4-62).

`PTPluginSetDeviceModes`
A pointer to the plug-in defined set device modes function (page 4-63).

Pointing Family Plug-in Header

You use the pointing family plug-in header, defined by the `PTPluginHeader` data type in the header field of the pointing family device dispatch table (page 4-20).

```
struct PTPluginHeader
{
    UInt32  version;    /* version of the plug-in interface */
    UInt32  reserved1;  /* reserved for use by Apple */
    UInt32  reserved2;  /* reserved for use by Apple */
    UInt32  reserved3;  /* reserved for use by Apple */
};

typedef struct PTPluginHeader PTPluginHeader;
```

Field descriptions

`version` An unsigned 32-bit integer that specifies the version of the pointing family plug-in interface to which the plug-in adheres. This version number is defined by the pointing

	family plug-in programming interface. Set this field to `kCoplandPTPluginVersion` (page 4-22).
`reserved1`	Reserved.
`reserved2`	Reserved.
`reserved3`	Reserved.

Pointing Device Plug-in Version

The pointing family plug-in version enumerator describes versions of the pointing family plug-in interface to which a plug-in might adhere. Currently, there is only one version. The version number appears in the `version` field of the pointing family plug-in header data structure, which is defined by the `PTPluginHeader` data type (page 4-21).

```
enum
{
    kCoplandPTPluginVersion =   0x0000001
};
```

Driver Description Data Structure

Each plug-in also must contain a plug-in description data structure, also called a driver description data structure, which is shown in Listing 4-1 (page 4-22). For more on this structure, see "Driver and Family Matching" (page 2-3).

Listing 4-1 Plug-In Driver Description Structure

```
DriverDescription TheDriverDescription =
{
    kDriverDescriptionSignature,
    kCoplandlDriverDescriptor,
    {
        "\pADB-3-01",
        1,0,0,0
    },
    {
        kDriverIsUnderExpertControl,
        "\pmouse",
```

```
        {0,0,0,0,0,0,0,0}
    },
    {
        1,
        {
            kServiceCategoryPointing,
            kNdrvTypeIsGeneric,
            0,0,0,0
        }
    }
};
```

- To indicate that your plug-in uses the pointing family plug-in interface, use the `kServiceCategoryPointing` constant.
- If a pointing family plug-in is an ADB device, you use an ADB match string (for instance, `ADB-3-01`) to describe it.
- `"\pmouse"` indicates that your plug-in controls a mouse and should be loaded into memory very early in the boot process.

For details on ADB match strings and their search order, see "ADB Family Reference" (page 3-5).

Pointing Family Plug-in Defined Function Types

This section describes the function pointer types defined by the pointing family plug-in programming interface.

PTPluginValidateHardwarePtr

Before the pointing family calls the initialize function provided by the plug-in (page 4-24), it calls the plug-in defined validate hardware function. The plug-in then confirms that the registered entry reference specified is the device that it knows how to control. (If it is not that device, the function sets the `isMyDevice` parameter to `false`.) The family uses this function to match plug-ins with devices.

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginValidateHardwarePtr) (RegEntryRef *device,
                    Boolean *isMyDevice);
```

For information about creating your own validate hardware function, see the description of the `MyPTPluginValidateHardwarePtr` function (page 4-55).

PTPluginInitializePtr

Once the pointing family matches a plug-in to a device, it calls the initialize function provided by the plug-in. The plug-in then fills out the device capabilities structure, which is defined by the `PTDeviceCapabilities` data type (page 4-14), and the device identifier structure, which is defined by the `PTDeviceIdentifier` data type (page 4-18), and performs any other initialization tasks.

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginInitializePtr)(
                    RegEntryRef *mouseRegistryEntryPtr
                    PTDeviceCapabilities *mouseCapabilities,
                    PTDeviceIdentifier *mouseIdentification;
```

For information about creating your own initialization function, see the description of the `MyPTPluginInitializePtr` function (page 4-56).

PTPluginTerminatePtr

When the pointing family discovers that a pointing device is no longer present (for example, a tablet is no longer attached), it calls the terminate function provided by the plug-in. The plug-in then performs any necessary clean-up operations, such as tearing down state, releasing memory, and so forth.

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginTerminatePtr)(void);
```

For information about creating your own terminate function, see the description of the `MyPTPluginTerminatePtr` function (page 4-58).

PTPluginStartIOPtr

When a client indicates via the `PTRegisterNewTracker` function (page 4-33) that it wants to use a pointing device, the pointing family calls the start I/O function provided by the plug-in. The plug-in then prepares for I/O operations in a device-specific fashion.

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginStartIOPtr)(void);
```

For information about creating your own start I/O function, see the description of the `MyPTPluginStartIOPtr` function (page 4-59).

PTPluginStopIOPtr

When a client indicates that it wants to discontinue the use of a pointing device, the pointing family calls the stop I/O function provided by the plug-in. The plug-in then terminates I/O operations in a device-specific fashion.

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginStopIOPtr)(void);
```

For information about creating your own stop I/O function, see the description of the `MyPTPluginStopIOPtr` function (page 4-60).

PTPluginGetNextDataPtr

When the pointing family needs the next piece of data to be passed to a client, it calls the get next data function, and the plug-in fills in the pointing device data structure, which is defined by the `PTData` data type (page 4-9).

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginGetNextDataPtr)(PTData *newData);
```

For information about creating your own get next data function, see the description of the `MyPTPluginGetNextDataPtr` function (page 4-61).

PTPluginGetDeviceModesPtr

When the client calls the `PTGetDeviceModes` function (page 4-36), the pointing family calls the plug-in defined get device modes function, and the plug-in fills in the pointing device modes structure, which is defined by the `PTDeviceModes` data type (page 4-11).

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginGetDeviceModesPtr) (ByteCount offset,
                    ByteCount numBytes, PTDeviceModes *modes);
```

For information about creating your own get device modes function, see the description of the `MyPTPluginGetDeviceModesPtr` function (page 4-62).

PTPluginSetDeviceModesPtr

When the client calls the `PTSetDeviceModes` function (page 4-35), the pointing family calls the plug-in defined set device modes function, and the plug-in changes the modes in the pointing device modes structure, which is defined by the `PTDeviceModes` data type (page 4-11).

The function pointer is defined by the pointing family as follows:

```
typedef OSStatus (*PTPluginSetDeviceModesPtr) (ByteCount offset,
                    ByteCount numBytes, PTDeviceModes *modes);
```

For information about creating your own set device modes function, see the description of the `MyPTPluginSetDeviceModesPtr` function (page 4-63).

Pointing Family Client Functions

This section describes the functions used by pointing family clients.Typical clients use pointing family functions to perform the following actions:

4. determine which pointing devices are available using the `PTGetNextDevice` function (page 4-28)
5. obtain information about the device capabilities and manufacturer via the `PTGetDeviceCapabilities` (page 4-29) and `PTGetDeviceIdentification` (page 4-31) functions in order to figure out which device is of interest
6. register interest in that device by calling the `PTRegisterNewTracker` function (page 4-33)
7. determine or set device modes information with the `PTSetDeviceModes` (page 4-35) and `PTGetDeviceModes` (page 4-36) functions
8. set the properties of a tracker using the functions `PTSetPosition` (page 4-47), `PTSetTrackerState` (page 4-44), `PTSetTrackerDataByOffset` (page 4-52), and `PTSetButtons` (page 4-50).
9. maintain tracker data using the `PTSetPinningRects` function (page 4-39)
10. obtain data from a tracker via the functions `PTGetTrackerData` (page 4-40), `PTGetTrackerDataByOffset`, `PTGetTrackerState` (page 4-43), `PTGetButtons` (page 4-49), and `PTSetButtons` (page 4-50).

Getting Information About Devices

Clients use the `PTGetNextDevice` function (page 4-28) to iterate through the list of available pointing devices. They use the functions `PTGetDeviceCapabilities` (page 4-29) and `PTGetDeviceIdentification` (page 4-31) to check the characteristics of each device they find to determine which, if any, they are interested in.

PTGetNextDevice

Retrieves the registered entry reference of the next device after the current device in the list of available pointing devices.

```
OSStatus PTGetNextDevice (
                    RegEntryRef *currentDevice, |
                    PTDeviceClass filter,
                    RegEntryRef **nextDevice);
```

`currentDevice` A pointer to a registry entry reference that identifies the current pointing device after which the next device is to be retrieved. If this parameter is `NULL`, `PTGetNextDevice` returns the first device in the list. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`filter` A pointing device class from the `PTDeviceClass` enumeration (page 4-15), which allows you to iterate through the list of available pointing devices, for instance, asking for all devices that are identified as mice. You can use any defined pointing family device class or you can use a device class defined by a plug-in. Use the `kAnyDeviceClass` enumerator (page 4-15) if you want the next device no matter what class it is.

`nextDevice` A pointer to a registered entry reference. On output, the `PTGetNextDevice` function provides a new registered entry reference that identifies the next device in the list of available pointing devices. If the current device specified in the `currentDevice` parameter is the last device in the list, `PTGetNextDevice` returns `NULL` in this parameter.

function result An operating system status code. If the client has passed in an unknown pointing device, `PTGetNextDevice` returns the result code `kPTUnknownRegEntryRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetNextDevice` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain the device capabilities for a pointing device, you can use the `PTGetDeviceCapabilities` function (page 4-29). To obtain unique device identification for a device, you can use the `PTGetDeviceIdentification` function (page 4-31).

PTGetDeviceCapabilities

Obtains the pointing device capabilities associated with a device.

```
OSStatus PTGetDeviceCapabilities (
                    RegEntryRef *device,
                    PTDeviceCapabilities *capabilities);
```

`device` A pointer to a registry entry reference returned by the `PTGetNextDevice` function (page 4-28). You set this registry entry reference to identify the pointing device whose capabilities are sought. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`capabilities` A pointer to a pointing device capabilities structure. On output, the `PTGetDeviceCapabilities` function provides the structure, which is defined by the `PTDeviceCapabilities` data type (page 4-14). The pointing device capabilities structure lists several capabilities associated with a device.

function result An operating system status code. If the client has passed in an unknown pointing device, `PTGetDeviceCapabilities` returns the result code `kPTUnknownRegEntryRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Once a client has identified a device via a registry entry reference, `PTGetDeviceCapabilities` returns a pointer to the device's associated device capabilities structure (or structures), defined by the `PTDeviceCapabilities` data type (page 4-14). Clients can then peruse the device's capacities.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetDeviceCapabilities` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To iterate through a list of available pointing devices, you can also use the `PTGetNextDevice` function (page 4-28). To obtain a unique pointing device identifier associated with a specific device, you can use the `PTGetDeviceIdentification` function (page 4-31).

PTGetDeviceIdentification

Gets the unique pointing device identifier associated with a device.

```
OSStatus PTGetDeviceIdentification (
                    RegEntryRef *device,
                    PTDeviceIdentifier *identification);
```

`device` A pointer to a registry entry reference returned by the `PTGetNextDevice` function (page 4-28). You set this registry entry reference to identify the pointing device whose identification is sought. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`identification`
A pointer to a pointing device identifier. On output, the `PTGetDeviceIdentification` function provides this value of type `PTDeviceIdentifier` (page 4-18), which describes a pointing device's unique identification, which has been defined by the manufacturer.

function result An operating system status code. If the client has passed in an unknown pointing device, `PTGetDeviceIdentification` returns the result code `kPTUnknownRegEntryRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Once a client has identified a device via a registry entry reference, `PTGetDeviceIdentification` returns a pointer to the device's associated pointing device identifier, defined by the `PTDeviceIdentifier` data type (page 4-18). Clients can then peruse the device's manufacturer information.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetDeviceIdentification` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To iterate through a list of available pointing devices, you can also use the `PTGetNextDevice` function (page 4-28). To retrieve a list of the capabilities associated with a specific pointing device, you can use the `PTGetDeviceCapabilities` function (page 4-29).

Registering With the Pointing Family

The `PTRegisterNewTracker` function (page 4-33) allows clients to indicate that they want to register and establish a connection with the pointing family.

Upon tracker registration, the pointing family provides two ways to obtain pointing data:

- *buffered data* for those clients who want to receive all their information from a specified tracker in a continuous stream. The pointing family keeps such data in a buffer so that when the client asks for the next piece of data (such as buffered mouse data), it is ready with the first piece of data from the buffer to pass to the client. In the `PTRegisterNewTracker` function (page 4-33), buffered data is indicated in the `bufferedData` parameter.
- *static data* for those clients who want the pointing family to add the tracker data together as it comes in and maintain the current state of the tracker. Such clients check with the pointing family to find out where the tracker is. In the `PTRegisterNewTracker` function (page 4-33), static data is indicated in the `stateData` parameter.

PTRegisterNewTracker

Registers a client to establish a connection with the pointing family.

```
OSStatus PTRegisterNewTracker (
                    RegEntryRef *device,
                    Boolean bufferedData,
                    Boolean stateData,
                    PTTrackerRef *tracker
                    ByteCount *dataSize);
```

`device` A pointer to a registry entry reference returned by the `PTGetNextDevice` function (page 4-28). This registry entry reference identifies the pointing device with which the client wants to establish a connection. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`bufferedData` A Boolean value that indicates whether the client wants the pointing family to buffer data. The client specifies `true` if buffered data is desired; otherwise, the client specifies `false`.

`stateData` A Boolean value that indicates whether the client wants the pointing family to provide state data. The client specifies `true` if state data is desired; otherwise, the client specifies `false`.

`tracker` A pointer to a pointing device tracker reference. On output, the `PTRegisterNewTracker` function provides a value of type `PTTrackerRef` (page 4-8) that represents the client's connection to the device. The client must use this connection in all future communications with the pointing family.

`dataSize` A pointer to a byte count. On output, the `PTRegisterNewTracker` function supplies the count that describes the size of the pointing data structure, defined by the `PTData` type (page 4-9), that the specified device generates.

function result An operating system status code. If the client has passed in an unknown pointing device, `PTRegisterNewTracker` returns the result code `kPTUnknownRegEntryRef`. If the specified device is not available or has already been registered for, the function returns the result code `kPointerFamilyError`. If the internal pointing family memory allocation has failed, the function returns the

result code `kPointerFamilyError`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Clients use `PTRegisterNewTracker` to indicate a desire to receive data from a specified device.

Note
In this developer release of Mac OS 8 only, once someone registers for a certain device, no one else can register for that device. In future developer releases of Mac OS 8, a more flexible scheme for sharing and unregistering devices will most likely be provided. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTRegisterNewTracker` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See "Registering With the Pointing Family" (page 4-32) for details on buffered and static data).

Setting and Retrieving Device Modes

Clients can use the `PTSetDeviceModes` (page 4-35) and `PTGetDeviceModes` (page 4-36) functions to set and retrieve device mode data, which is the kind of information that pointing family clients pass back to pointing device, usually to

set a mode on the device. Examples of device modes include tactile feedback for steering wheels, absolute versus relative mode for tablets, and LED displays.

Note
Because the modes available vary between device classes and devices, clients must understand the mode data structure of the device they are working with. ▲

PTSetDeviceModes

Sets the modes on a device.

```
OSStatus PTSetDeviceModes
                    (RegEntryRef *device,
                    ByteCount offset,
                    ByteCount numBytes
                    PTDeviceModes *modes);
```

`device` A pointer to a registry entry reference returned by the `PTGetNextDevice` function (page 4-28). You set this registry entry reference to identify the pointing device whose device modes are to be set. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`offset` A byte count that describes the offset into the device mode structure (page 4-11) of the device mode you want to set.

`numBytes` A byte count that describes the number of bytes of device mode data to be updated.

`modes` A pointer to a pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11). This structure contains the new device mode data as indicated by the `offset` and `numBytes` parameters.

function result An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client has passed in an unknown device, the function returns

`kPTUnknownRegEntryRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

When the client calls `PTSetDeviceModes`, the pointing family calls the `MyPTPluginSetDeviceModesPtr` function (page 4-63) since the plug-in keeps this information.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetDeviceModes` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To retrieve the device modes for a given device, you can use the `PTGetDeviceModes` function (page 4-36). For more on device modes, see "Pointing Device Modes Structure" (page 4-11) and "Setting and Retrieving Device Modes" (page 4-34).

PTGetDeviceModes

Retrieves the mode on a device.

```
OSStatus PTGetDeviceModes (
                    RegEntryRef *device,
                    ByteCount offset,
                    ByteCount numBytes
                    PTDeviceModes *modes);
```

`device`	A pointer to a registry entry reference returned by the `PTGetNextDevice` function (page 4-28). You set this registry entry reference to identify the device whose device mode you want to retrieve. For more on registry entry references, see "About the I/O Architecture" (page 1-3).
`offset`	A byte count that describes the offset into the pointing device modes structure (page 4-11) of the device mode data you want to get.
`numBytes`	A byte count that describes the number of bytes of data you want, beginning at the offset.
`modes`	A pointer to a pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11). On output, the family fills in the structure with the amount of data specified in the `numBytes` parameter, beginning at the offset indicated in the `offset` parameter.
function result	An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client has passed in an unknown device, the function returns `kPTUnknownRegEntryRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

When the client calls `PTGetDeviceModes`, the pointing family calls the `MyPTPluginGetDeviceModesPtr` function (page 4-62) since the plug-in keeps this information.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetDeviceModes` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set the device mode for a given device, you can use the `PTSetDeviceModes` function (page 4-35). For more on device modes, see "Pointing Device Modes Structure" (page 4-11) and "Setting and Retrieving Device Modes" (page 4-34).

Maintaining Trackers

In order for clients to instruct the pointing family to pin a tracker's static position, they must provide **pinning rectangles**. Such rectangles restrict a tracker's position to within their boundaries, no matter how far the device moves. For example, the system cursor always stays inside the pinning rectangles defined by the main screen.

Note
In this developer release of Mac OS 8 only, there is support for one pinning rectangle per tracker. In subsequent developer releases of Mac OS 8, multiple pinning rectangles will most likely be supported ◆

The graphics system (for instance, QuickDraw) sets the pinning rectangles on the system cursor's tracker. Other clients may want to use pinning rectangles for other purposes, for instance, to pin the tracker inside a window. If a tracker has no pinning rectangles, the pointing family does not pin its position.

PTSetPinningRects

Sets pinning rectangles for a tracker.

```
OSStatus PTSetPinningRects (
                    PTTrackerRef tracker,
                    PTPinningRectList *rectList);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose pinning rectangles you want to set.

`rectList` A pointer to a list of pinning rectangles. This list is described by the `PTPinningRectList` (page 4-18) structure. To stop pinning, clients can set this parameter to `NULL`.

Note
In this developer release of Mac OS 8, there can only be one pinning rectangle in the list. ◆

function result An operating system status code. If the pointing family internal memory allocation has failed, `PTSetPinningRects` returns the `kPointerFamilyError` result code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

`PTSetPinningRects` sets the pinning rectangles for a tracker to those pointed to by the `rectList` parameter.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetPinningRects` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For details on pinning rectangles, see "Maintaining Trackers" (page 4-38).

Getting Tracker-Buffered Data

Clients can obtain data from a pointing family tracker by retrieving the next piece of pointing family information from the buffer using the `PTGetTrackerData` function (page 4-40) or by clearing the tracker's buffer and starting fresh using the `PTFlushTrackerBuffer` function (page 4-42).

PTGetTrackerData

Retrieves the next piece of pointing data from a tracker's buffer.

```
OSStatus PTGetTrackerData (
                    PTTrackerRef tracker,
                    ByteCount dataSize
                    PTData *dataPtr, );
```

`tracker` A pointing family tracker reference returned by the `PTRegisterNewTracker` function (page 4-33). This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose buffer pointing data is being retrieved.

`dataSize`	A byte count that describes the size of the data in the pointing data structure pointed to by the `dataPtr` parameter. This size is returned in the `dataSize` parameter of the `PTRegisterNewTracker` function (page 4-33). Because the size of the pointing data structure is variable, the client needs to inform the pointing family of the data's expected size.
`dataPtr`	A pointer to the pointing data structure. On output, the `PTGetTrackerData` function provides the structure defined by the `PTData` (page 4-9) type describes the pointing data that has been retrieved. The client must allocate a structure big enough to hold the amount of data specified in the `dataSize` parameter and pass a pointer to that memory in this parameter.
function result	An operating system status code. If the pointing family's internal memory allocation failed, `PTGetTrackerData` returns the result code `kPTMemoryAllocationFailed`. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetTrackerData` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For details on obtaining tracker data, see "Getting Tracker-Buffered Data" (page 4-40).

To clear out a buffer so you can receive all tracker data from that point on, use the `PTFlushTrackerBuffer` function (page 4-42).

PTFlushTrackerBuffer

Flushes the buffer associated with a tracker.

```
OSStatus PTFlushTrackerBuffer (PTTrackerRef tracker);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose buffer data is being flushed.

function result An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

As a pointing device moves, a tracker's buffer fills up. A client may want to ask for all the information from the present time forward. You can use `PTFlushTrackerBuffer` to clear out a tracker's buffer.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTFlushTrackerBuffer` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For details on obtaining tracker data, see "Getting Tracker-Buffered Data" (page 4-40).

To remove the next piece of data from a tracker's buffer, you can use the `PTGetTrackerData` function (page 4-40).

Checking Tracker State

This section describes the functions clients can use to check and change the state of a pointing family tracker: `PTGetTrackerState` (page 4-43), `PTGetPosition` (page 4-46), `PTGetButtons` (page 4-49), and `PTGetTrackerDataByOffset` (page 4-52). In these functions, the pointing family maintains state, accumulates position, and replaces all other fields (such as buttons, device-specific data) every time new data comes from the device.

PTGetTrackerState

Obtains the current state of a tracker.

```
OSStatus PTGetTrackerState
                    (PTTrackerRef tracker,
                    ByteCount dataSize
                    PTData *data);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose current state you want to retrieve.

`dataSize` A byte count that describes the size of the data in the pointing data structure pointed to by the `data` parameter. This size is returned in the `dataSize` parameter of the `PTRegisterNewTracker` function (page 4-33). Because the size of the pointing data structure is variable, the client needs to inform the pointing family of the data's expected size.

`data` A pointer to the pointing data structure. On output, the `PTGetTrackerState` function provides the structure, defined by the `PTData` (page 4-9) type, describes the state of the specified tracker including its buttons, position, and any device-specific data. The client must allocate a structure big enough to hold the amount of data specified in the `dataSize` parameter and pass a pointer to that memory in this parameter.

function result An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client

has specified an invalid pointing family tracker reference, the function returns the result code `kPTInvalidTrackerRef`. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Most application clients only need to obtain a tracker's state and do not need to set or change this information. However, some clients may want to initialize and set their tracker's state.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetTrackerState` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For details on tracker states, see "Checking Tracker State" (page 4-43). To set the state of a tracker, you can use the `PTSetTrackerState` function (page 4-44).

PTSetTrackerState

Sets the state of a tracker.

```
OSStatus PTSetTrackerState (
                    PTTrackerRef tracker,
                    ByteCount dataSize,
                    PTData *data);
```

`tracker`	A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose current state you want to set.
`dataSize`	A byte count that describes the size of the data in the pointing data structure (page 4-9) pointed to by the `data` parameter.
`data`	On input, a pointer to the pointing data structure. This structure, defined by the `PTData` (page 4-9) type, describes the state of the specified tracker.
function result	An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetTrackerState` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain the state of a tracker, you can use the `PTGetTrackerState` function (page 4-43).

Working WithTracker Position

The `PTGetPosition` (page 4-46) and `PTSetPosition` (page 4-47) functions described in this section are subsets of the `PTGetTrackerState` (page 4-43) and `PTSetTrackerState` (page 4-44) functions.

PTGetPosition

Retrieves a tracker's position.

```
OSStatus PTGetPosition  (
                    PTTrackerRef tracker,
                    PTPosition *position);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose position you want to obtain.

`position` A pointer to a position structure. On output, the `PTGetPosition` function provides this structure, defined by the `PTPosition` (page 4-10) type, to describe the position of the tracker.

function result An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Instead of allocating an entire pointing data structure, the `PTGetPosition` function lets you obtain a tracker's position separately.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetPosition` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set a tracker's position, you can use the `PTSetPosition` function (page 4-47). To move a tracker's position, you can use the `PTMovePosition` function (page 4-48).

PTSetPosition

Replaces a tracker's current position with a specified position.

```
OSStatus PTSetPosition (
                    PTTrackerRef tracker,
                    PTPosition *position);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose position you want to set.

`position` A pointer to a pointing position structure. This structure, defined by the `PTPosition` (page 4-10) data type, describes the new position of the tracker.

function result An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Instead of passing an entire pointing data structure, the `PTSetPosition` function lets you manipulate a tracker's position separately.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetPosition` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain a tracker's position, you can use the `PTGetPosition` function (page 4-46). To move a tracker's position, you can use the `PTMovePosition` function (page 4-48).

PTMovePosition

Adds a specified position to an existing position.

```
OSStatus PTMovePosition (
                    PTTrackerRef tracker,
                    PTPosition *position);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose position you want to move.

`position` A pointer to a pointing position structure. This structure, defined by the `PTPosition` (page 4-10) data type describes a position to be added to the existing tracker position.

function result An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTMovePosition` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set a tracker's position, you can use the `PTSetPosition` function (page 4-47). To get a tracker's position, you can use the `PTGetPosition` function (page 4-46).

Working With Tracker Buttons

The `PTGetButtons` (page 4-49) and `PTSetButtons` (page 4-50) functions described in this section are subsets of the `PTGetTrackerState` (page 4-43) and `PTSetTrackerState` (page 4-44) functions.

PTGetButtons

Obtains the button state of a tracker.

```
OSStatus PTGetButtons (
                    PTTrackerRef tracker,
                    PTButtonState *buttons);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose buttons you want to obtain.

position A pointer to a button state. On output, the `PTGetButtons` function supplies the `PTButtonState` (page 4-11) data type, which describes the button state of the tracker.

function result An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Instead of getting an entire pointing data structure, the `PTGetButtons` function lets you obtain information about a tracker's buttons separately.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetButtons` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set the button state of a tracker, you can use the `PTSetButtons` function (page 4-50).

PTSetButtons

Sets the button state of a tracker.

```
OSStatus PTSetButtons (
                    PTTrackerRef tracker,
                    PTButtonState buttons);
```

`tracker`	A pointing device tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose buttons you want to set.
`position`	A button state structure. The `PTButtonState` (page 4-11) data type describes the desired button state of the tracker.
function result	An operating system status code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

Instead of passing an entire pointing data structure, the `PTSetButtons` function lets you manipulate a tracker's buttons separately.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetButtons` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To get the button state of a tracker, you can use the `PTGetButtons` function (page 4-49). For a discussion of the relationship of this function to be `PTGetTrackerState` (page 4-43) and `PTSetTrackerState` (page 4-44) functions, see "Checking Tracker State" (page 4-43).

Getting and Setting Tracker Data By Offset

This section describes the `PTGetTrackerDataByOffset` (page 4-52) and `PTSetTrackerDataByOffset` (page 4-53) functions, subsets of the `PTGetTrackerState` (page 4-43) and `PTSetTrackerState` (page 4-44) functions. `PTGetTrackerDataByOffset` and `PTSetTrackerDataByOffset` are designed to allow clients to get and set any subset of the pointing data structure, which is defined by the `PTData` type (page 4-9).

PTGetTrackerDataByOffset

Obtains any subset of a tracker's state.

```
OSStatus PTGetTrackerDataByOffset (
                    PTTrackerRef tracker,
                    ByteCount offset,
                    ByteCount numBytes,
                    void *buffer);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose data you want to access.

`offset` A byte count that describes the offset into a pointing data structure, defined by the `PTData` type (page 4-9), of the data you want to obtain.

`numBytes` A byte count of the data you want to access.

`buffer` A pointer to a buffer where the family, on output, is to put the amount of data specified in the `numBytes` parameter.

function result An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTGetTrackerDataByOffset` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To set the individual fields of data associated with a tracker, you can use the `PTSetTrackerDataByOffset` function (page 4-53).

PTSetTrackerDataByOffset

Sets individual fields of data associated with a tracker.

```
OSStatus PTSetTrackerDataByOffset (
                    PTTrackerRef tracker,
                    ByteCount offset,
                    ByteCount numBytes,
                    void *buffer);
```

`tracker` A pointing family tracker reference. This value of type `PTTrackerRef` (page 4-8) specifies the tracker whose data you want to set.

`offset` A byte count that describes the offset into a pointing data structure, defined by the `PTData` type (page 4-9), of the data you want to set.

`numBytes` A byte count that describes the number of bytes of data.

`buffer` A pointer to a buffer containing the amount of data specified in the `numBytes` parameter.

function result An operating system status code. If the internal memory allocation for the pointing family has failed, the function returns the `kPTMemoryAllocationFailed` result code. If the client has specified an invalid pointing family tracker reference, the function returns the `kPTInvalidTrackerRef` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

The `PTSetTrackerDataByOffset` function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To retrieve the individual fields of data associated with a tracker, you can use the `PTGetTrackerDataByOffset` function (page 4-52).

Pointing Family Plug-In-Defined Functions

Any pointing family plug-in must implement these calls for the pointing family to use. The pointing family provides a flexible interface which allows plug-ins to generate, and clients to receive, device-specific data.

Note
A driver developer may develop a separate library that exports a set of functions more specific to its data and easier for clients to use. See "About the I/O Architecture" (page 1-3) for details. ◆

Validating Pointing Devices

When the I/O system discovers a pointing device, the pointing family must determine which plug-in goes with the device. It does this by calling the `MyPTPluginValidateHardwarePtr` function (page 4-55) for each possible plug-in until one of the plug-ins indicates it owns the device.

MyPTPluginValidateHardwarePtr

Instructs the plug-in to indicate whether or not the pointing device specified by the registry entry reference is the piece of hardware expected by the plug-in.

```
OSStatus MyPTPluginValidateHardwarePtr
                    (RegEntryRef *device,
                    Boolean *isMyDevice);
```

`device` A pointer to a registry entry reference, defined by the `RegEntryRef` data type. This reference identifies the device to be tested. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`isMyDevice` A pointer to a Boolean value. On output, the plug-in sets this parameter to `true` to indicate that the pointing device identified by the registry entry reference in the `device` parameter belongs to the plug-in. Otherwise, the plug-in sets this parameter to `false`.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its validate hardware function has been successful. If an error occurs, it should return the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

A plug-in should attempt to talk to its device during this call. However, the device may not be there any longer. The plug-in should do whatever else is necessary to determine if the device belongs to it.

Within `MyPTPluginValidateHardwarePtr`, the plug-in must:

1. Set up any state required to execute the validate hardware function.
2. Test to see if it owns the device. If the device is an ADB device, the plug-in must read from the device to make sure it is actually present. If the read times out, the plug-in must return `false` to the pointing family. If the device is a virtual device (that is, a plug-in that is not actually associated with the hardware), the plug-in must return `true`.
3. Tear down all state, since the plug-in is may be unloaded.

The pointing family then loads and calls the initialize function for the plug-in associated with the specified device.

The `PTPluginValidateHardwarePtr` type (page 4-23) defines a pointing family plug-in's validate hardware function.

SEE ALSO

For general information about implementing your own plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

Initializing and Terminating Plug-ins

The pointing family calls the `MyPTPluginInitializePtr` function (page 4-56) as soon as a device is matched to a plug-in and the `MyPTPluginTerminatePtr` function (page 4-58) when it receives notification that the device is no longer present.

MyPTPluginInitializePtr

Instructs the plug-in to fill out the device capabilities data structure, defined by the `PTDeviceCapabilities` data type (page 4-14), and the pointing device

identifier structure, defined by the `PTDeviceIdentifier` data type (page 4-18), and then performs any necessary initialization operations.

```
OSStatus MyPTPluginInitializePtr (
                    RegEntryRef *device,
                    PTDeviceCapabilities *deviceCapabilities,
                    PTDeviceIdentifier *deviceIdentification);
```

`device` A pointer to a registry entry reference, defined by the `RegEntryRef` data type. This reference identifies the device to be initialized. For more on registry entry references, see "About the I/O Architecture" (page 1-3).

`deviceCapabilities`
A pointer to a pointing device capabilities structure, which is defined by the `PTDeviceCapabilities` data type (page 4-14).

`deviceIdentification`
A pointer to a pointing device identifier structure, defined by the `PTDeviceIdentifier` data type (page 4-18). This structure identifies the brand and model of the pointing device identified in the `device` parameter.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its initialize function has been successful. If an error occurs, it should return the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The `MyPTPluginInitializePtr` type (page 4-24) defines a pointing family plug-in's initialize function.

SEE ALSO

To terminate a plug-in, the pointing family calls the `MyPTPluginTerminatePtr` function (page 4-58).

For general information about implementing your own plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

MyPTPluginTerminatePtr

Instructs the plug-in to perform necessary clean-up operations.

```
OSStatus MyPTPluginTerminatePtr (void);
```

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its terminate function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The pointing family calls the `MyPTPluginTerminate` function when it finds out the device is no longer present.

The `MyPTPluginTerminatePtr` type (page 4-24) defines a pointing family plug-in's terminate function.

SEE ALSO

To initialize a plug-in, the pointing family calls the `MyPTPluginInitializePtr` function (page 4-24).

For general information about implementing plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

Controlling Device I/O

The pointing family uses the `MyPTPluginStartIOPtr` function(page 4-59) to notify the plug-in that it wants to start getting data. When the pointing family no longer needs to gather data from a device, it uses the `MyPTPluginStopIOPtr` function (page 4-60) to notify the plug-in to perform the corresponding cleanup.

MyPTPluginStartIOPtr

Instructs the plug-in to perform necessary preparations for I/O operations.

```
OSStatus MyPTPluginStartIOPtr (void);
```

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its start I/O function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The pointing family calls the `MyPTPluginStartIOPtr` function when a client registers interest in a pointing device and its information. A mouse or another ADB plug-in typically would then call `ADBOpen` as part of their implementation of the start I/O function. For details on the `ADBOpen` function, see "ADB Family Reference" (page 3-5).

The `MyPTPluginStartIOPtr` type (page 4-25) defines a pointing family plug-in's start I/O function.

SEE ALSO

For a general discussion of creating plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

For more on notifying the plug-in that data is forthcoming, see "Controlling Device I/O" (page 4-58).

To notify the plug-in that it no longer wants data from a device, the pointing family calls `MyPTPluginStopIOPtr` (page 4-60).

MyPTPluginStopIOPtr

Instructs the plug-in to perform appropriate clean-up operations needed when the pointing family has stopped gathering data.

```
OSStatus MyPTPluginStopIOPtr (void);
```

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its stop I/O function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The pointing family calls `MyPTPluginStopIOPtr` when clients indicate that they no longer want a connection and are no longer interested in the plug-in's data. For example, an ADB device would call the `ADBClose` function here. For details on the ADB client functions, see "ADB Family Reference" (page 3-5).

However, the plug-in does not need to dismantle all state data. Another client may want data, so using this function saves the plug-in the trouble of reinitializing.

The `MyPTPluginStopIOPtr` type (page 4-25) defines a pointing family plug-in's stop I/O function.

SEE ALSO

For details on the `ADBClose` function, see "ADB Family Reference" (page 3-5).

For a general discussion of creating plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

For more on notifying the plug-in that data has stopped, see "Controlling Device I/O" (page 4-58).

To notify the plug-in to that a device has been discovered, the pointing family calls `MyPTPluginStartIOPtr` (page 4-59).

Getting Device Data

The pointing family calls the `MyPTPluginGetNextDataPtr` function (page 4-61) every time a client wants another piece of data. The plug-in gets the data from the device and manipulates it to fit into the pointing data structure, defined by the `PTData` type (page 4-9).

MyPTPluginGetNextDataPtr

Instructs the plug-in to fill in data that client is awaiting.

```
OSStatus MyPTPluginGetNextDataPtr (
                    PTData *newData);
```

`newData` A pointer to the pointing data structure defined by the `PTData` data type (page 4-9). On output, the plug-in fills in the structure.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its get next data function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

`MyPTPluginGetNextDataPtr` blocks until the device has data available and can fill in the `theMouseData` parameter with the new information.

The `PTPluginGetNextDataPtr` type (page 4-25) defines a pointing family plug-in's get next data function.

SEE ALSO

For a general discussion of creating plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

Setting and Getting Device Modes

MyPTPluginGetDeviceModesPtr

Instructs the plug-in to retrieve the contents of the pointing device modes structure.

```
OSStatus MyPTPluginGetDeviceModesPtr (
                    ByteCount offset,
                    ByteCount numBytes,
                    PTDeviceModes *modes);
```

`offset` A byte count that describes the offset into the pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that the plug-in is to retrieve.

`numBytes` A byte count that describes the number of bytes (starting at the offset specified in the `offset` parameter) of expected data in the device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that the plug-in is to retrieve.

`modes` A pointer to a pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that the family has allocated for the plug-in, on output, to fill in with the amount of data specified in the `numBytes` parameter.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its get device modes function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The `PTPluginGetDeviceModesPtr` type (page 4-26) defines a pointing family plug-in's get device modes function.

SEE ALSO

For a general discussion of creating plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

To instruct a plug-in to change the contents of the pointing device modes structure, the pointing family calls the `MyPTPluginSetDeviceModes` function (page 4-63).

MyPTPluginSetDeviceModesPtr

Instructs the plug-in to change the contents of its pointing device modes structure.

```
OSStatus MyPTPluginGetDeviceModesPtr (
                    ByteCount offset,
                    ByteCount numBytes,
                    PTDeviceModes *modes);
```

`offset` A byte count that describes the offset into the pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that the plug-in is to set.

`numBytes` A byte count that describes the number of bytes of expected data in the device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that the plug-in is to replace.

`modes` A pointer to a pointing device modes structure, defined by the `PTDeviceModes` data type (page 4-11), that contains the new mode data, starting at the offset into the pointing modes structure specified in the `offset` parameter.

function result An operating system status code. Your plug-in should return the result code `noErr` to indicate that its set device modes function has been successful. If an error occurs, it should return a the `kPointerFamilyError` result code. See "Pointing Family Result Codes" (page 4-64) for a list of the result codes the pointing family can return.

DISCUSSION

The `PTPluginGetDeviceModesPtr` type (page 4-26) defines a pointing family plug-in's get device modes function.

SEE ALSO

For a general discussion of creating plug-in defined functions, see "Pointing Family Plug-In-Defined Functions" (page 4-54).

To instruct a plug-in to fill out the pointing device modes structure, the pointing family calls the `MyPTPluginSetDeviceModes` function (page 4-63).

Pointing Family Result Codes

Many pointing family functions return result codes. The various result codes specific to the pointing family are listed here.

`kPointerFamilyError`	-1	Pointing family error
`kPTUnknownRegEntryRef`	-2	Registry entry reference is unknown.
`kPTInvalidTrackerRef`	-3	Tracker reference is invalid.
`KPTMemoryAllocationFailed`	-4	Memory allocation has failed.

Glossary

absolute data Position information sent to the pointing family by a plug-in that generates the actual coordinates of a pointing device, for instance (10, 20).

device mode data The kind of information that pointing family clients pass to a pointing device, usually to set a mode on the device. Examples of device modes include tactile feedback for steering wheels, absolute versus relative mode for tablets, and LED displays. The pointing family supports device mode data with the pointing device modes structure, which is defined by the `PTDeviceModes` data type (page 4-11).

latency The time it takes a plug-in to gather up data generated by the device and send it to the pointing family (in microseconds).

pinning rectangles Rectangles that restrict a position to within their boundaries, no matter how far a device may move. A tracker's static position does not extend beyond a pinning rectangle. If a tracker has more than 1 pinning rectangle, the position can be inside any of them, but not outside all of them. When a pointing device sends data that is outside the tracker's pinning rectangles in 1 dimension, that dimension does not change.

pointing devices User-input mechanisms (such as mice, tablets, and joysticks) that indicate position and orientation and facilitate movement through the user interface. Pointing devices are commonly used to control cursors and to view objects in space.

pointing family That element of the I/O system that provides support for pointing devices in Mac OS 8. The pointing family distributes data from pointing devices to application clients (for instance, graphics and paint applications, games, and device control panels) and to system software. It provides clients with a common interface to all pointing devices. Furthermore, the pointing family applies standard manipulations of pointing data before providing data to clients.

pointing family clients Software that wants to obtain data from pointing devices, including applications such as graphics and paint programs as well as games that often take advantage of the special capabilities of pointing devices. Control panel applications sometimes specify the behavior of pointing devices. Finally, system software, for instance the Apple Event Manager and the graphics system, such as QuickDraw, may want to obtain data from pointing devices.

pointing family plug-ins Software modules, also called drivers, that get data from the pointing devices themselves (such as mice, tablets, joysticks, and 3D trackballs) and pass it up to the pointing family.

pointing family trackers Representations of connections between devices and clients of the pointing family. Each tracker stores information about what kind of information the client wants and how the client would like to receive that data.

relative data Position information sent to the pointing family by a plug-in that describes how far the pointing device moved from an already established coordinate, for instance, if the coordinate was (10, 20), it might have changed to (11, 21). In this case, the data (1,1) would be generated.

PCI Family Reference

Contents

Constants and Data Types

PCI Assigned-Address Property Structure

This structure is used for accessing the PCI assigned-address property.

```
struct PCIAssignedAddress {
    PCIAddressSpaceFlags   addressSpaceFlags;
    PCIBusNumber           busNumber;
    PCIDeviceFunction      deviceFunctionNumber;
    PCIRegisterNumber      registerNumber;
    UnsignedWide           address;
    UnsignedWide           size;
};
typedef PCIAssignedAddress *PCIAssignedAddressPtr;

typedef struct PCIAssignedAddress PCIAssignedAddress;
```

Field descriptions

`addressSpaceFlags` The I/O information common to all devices.

`busNumber` The number that identifies the bus. This is a value in the 0 through 255 range.

`deviceFunctionNumber` The number that identifies the function.It can be a number in the range 0 through 7.

`registerNumber` The number that identifies the configuration register number.

`address` Physical base address for adress space.

`size` Size of the address space.

PCI Address Space Flags

The PCI family provides the `PCIAddressSpaceFlags` data type and enumerated values for defining PCI address space. A value of type `PCIAddressSpaceFlags`is part of the the `PCIAssignedAddress` structure (page 5-5).

```
enum {
     kPCIRelocatableSpace  = 0x80,
     kPCIPrefetchableSpace = 0x40,
     kPCIAliasedSpace      = 0x20,
     kPCIAddressTypeCodeMask= 0x03,
     kPCIConfigSpace       = 0,
     kPCIIOSpace           = 1,
     kPCI32BitMemorySpace  = 2,
     kPCI64BitMemorySpace  = 3
};
typedef UInt8 PCIAddressSpaceFlags;
```

Enumerator descriptions

`kPCIRelocatableSpace`
: The physically accessible memory space may be relocated within defined memory space or I/O space.

`kPCIPrefetchableSpace`
: The address space can be read ahead in a FIFO scheme without disturbing the operation of the device.

`kPCIAliasedSpace`
: The address space can be duplicated an readdressed.

`kPCIAddressTypeCodeMask`
: The address space can be identified as one of four region types.

`kPCIConfigSpace`
: The address space is Configuration space. This value fits in `kPCIAddressTypeCodeMask`

`kPCIIOSpace`
: The address space is I/O space. This value fits in `kPCIAddressTypeCodeMask`.

`kPCI32BitMemorySpace`
: The address space is 32-bit memory space. This value fits in `kPCIAddressTypeCodeMask`.

`kPCI64BitMemorySpace`
: The address space is 64-bit memory space. This value fits in `kPCIAddressTypeCodeMask`.

PCIDeviceFunction

The PCI family defines the `PCIDeviceFunction` data type and enumerated values for device function types. The `PCIDeviceFunction` type is used in the structure type `PCIAssignedAddress (page 5-5)`.

```
enum {
    kPCIDeviceNumberMask= 0x1F,
    kPCIFunctionNumberMask= 0x07
};

typedef UInt8 PCIDeviceFunction;
```

Enumerator descriptions

`kPCIDeviceNumberMask`
: The bit mask field that identifies a device. Each bus can support a maximum of 32 devices.

`kPCIFunctionNumberMask`
: The bit mask field that identifies the function. It can be a value 0 through 7.

PCIBusNumber

The PCI family defines the `PCIBusNumber` data type. This data type defines the number of a specific PCI bus. The bus can be one of 256 architectural buses. The `PCIBusNumber` type is used in the structure type `PCIAssignedAddress (page 5-5)`.

```
typedef UInt8 PCIBusNumber;
```

PCIRegisterNumber

The PCI family defines the `PCIRegisterNumber` data type. This data type defines the number of a specific configuration register. The register can be one of 256 registers. The `PCIRegisterNumber` type is used in the structure type `PCIAssignedAddress (page 5-5)`

```
typedef UInt8 PCIRegisterNumber;
```

PCIConfigAddress

The PCI family defines the `PCIConfigAddress` data type. This data type defines the offset into the configuration space registers. This register is used when the driver needs to access configuration space. The `PCIConfigAddress` type is used in the configuration space cycle functions described in "PCI Kernel Cycle Routines" (page 5-23) and the `PCIConfigAddressGetDeviceData` function (page 5-46).

```
typedef LogicalAddress PCIConfigAddress;
```

PCIIOAddress

The PCI family defines the `PCIIOAddress` data type. This data type defines the offset into the PCI I/O space. This register is used when the driver needs to accessPCI I/O space. The `PCIIOAddress` type is used in the I/O space cycle functions described in "PCI Kernel Cycle Routines" (page 5-23).

```
typedef LogicalAddress PCIIOAddress;
```

PCIIOIteratorData Structure

The PCI family defines the `PCIIOIteratorData` data type to provide information about the devices known to the PCI family.

```
struct PCIIOIteratorData {
IOCommonInfo IOCI;
char Name[32];
UInt32 Domain;
UInt32 BusNumber;
UInt32 ConfigAddress;
};

typedef struct PCIIOIteratorData PCIIOIteratorData;
```

Field descriptions

`IOCI`	The I/O information common to all devices.
`Name`	The size of the store, expressed in bytes.
`Domain`	The number that identifies an electrically separate PCI bus. A domain can support up to 256 buses.
`BusNumber`	The number that identifies the bus. This is a value in the 0 through 255 range.
`ConfigAddress`	Phyical base address for Configuration space.

PCI Plugin Header

```
struct PCIPluginHeader {
     UInt32              version;
     UInt32              reserved1;
     UInt32              reserved2;
     UInt32              reserved3;
     PluginLoadID        thisPluginLoadID;
};
typedef PCIPluginHeader *PCIPluginHeaderPtr;

typedef struct PCIPluginHeader PCIPluginHeader;
```

Field descriptions

`version`	The interface version. This value must be statically initialized with the PCI header interface version (page 5-12).
`reserved1`	Reserved for future use.
`reserved2`	Reserved for future use.
`reserved3`	Reserved for future use.
`thisPluginLoadId`	The load ID for the plug-in that has been loaded and initialized. The PCI family stores the load ID for the plug-in in this field.

PCI Bridge Descriptor

The PCI bridge descriptor definition defines the template that should be used when developing a driver to be plugged-in to the PCI family. The table that uses this structure must be exported and it must be called the PluginDispatchTable for the PCI family.

```
struct PCIBridgeDescriptor {
     PCIPluginHeader          InterfaceHeader;
     DriverDescription *      TheDomainDriverDescription;
     InitializeFuncPtr        InitializeFunc;
     DefaultEnablerFuncPtr    DefaultBridgeEnablerFunc;
     DefaultDisablerFuncPtr   DefaultBridgeDisablerFunc;
     DefaultDispatcherFuncPtr DefaultBridgeDispatcherFunc;
     FinalizeFuncPtr          FinalizeFunc;
};
typedef PCIBridgeDescriptor *PCIBridgeDescriptorPtr;

typedef struct PCIBridgeDescriptor PCIBridgeDescriptor;
```

Field descriptions

`InterfaceHeader`	The interface version. This value must be statically initialized with the PCI header interface version (page 5-12).
`TheDomainDriverDescription`	The defined structure used by the driver family.

`InitializeFunc` Function called to initialize the device and bring it to a known state. See `PCIBridgePluginInitialize` function (page 5-68).

`DefaultBridgeEnablerFunc`
Function called to invoke the bridge interrupt enabler function. See the `DefaultBridgeEnabler` function (page 5-69)

`DefaultBridgeDisablerFunc`
Function called to invoke the bridge interrupt disabler function. See the `DefaultBridgeDisabler` function (page 5-70)

`DefaultBridgeDispatcherFunc`
Function called to invoke the transversal interrupt service routine. See the `DefaultBridgeDispatcher` function (page 5-71)

`FinalizeFunc` Function called to shut down plug-in bridge devices.t See `PCIBridgePluginFinalize` function (page 5-72).

PCI Bridge Variables

The PCI family defines the `DefaultBridgeVariables` data type to be used for interrupt dispatching. The interrupt handling is provided by a bridge plug-in or multifunction plug-in.

```
struct DefaultBridgeVariables {
     UInt32                  lastEntryIntCount;
     InterruptMemberNumber   lastServicedMember;
     UInt32                  totalMembersScanned;
     UInt32                  totalMemberCount;
     UInt8 *                 memberEnableFlags;
};
typedef DefaultBridgeVariables *DefaultBridgeVariablesPtr;
```

```
typedef struct DefaultBridgeVariables DefaultBridgeVariables;
```

Field descriptions

`lastEntryIntCount` To be provided later.

`lastServiceMember` To be provided later.

`totalMembersScanned` To be provided later.

`totalMemberCount` To be provided later.

`memberEnableFlags` To be provided later.

PCI Header Interface Version

The PCI family defines a constant for tracking the PCI family interface release. The plug-in will use this value to indicate the functionality that it supports. As versions of the PCI family are released, the number of enumerated values will increase. Currently, only one release has been made and therefore, this enumerator indicates that this is the initiali release.

```
enum {
kPCIPluginVersion1000= 0x01000000
};
```

PCI Error Codes

Not available at this time.

PCI Reg Property Structure

The PCI family defines `PCIRegProperty structure as` subordinate PCI device tree 'reg' property structure definition. It is used in conjunction with the Name Registry property found in `RegEntryRef`. For more information about the “reg” property, refer to the IEEE 1275-1994 specification on PCI Bus Binding.

```
struct PCIRegProperty {
     UInt32         physicalHigh;
     UInt32         physicalMiddle;
```

```
    UInt32          physicalLow;
    UInt32          propAddress;
    UInt32          propLength;
};
typedef PCIRegProperty *PCIRegPropertyPtr;

typedef struct PCIRegProperty PCIRegProperty;
```

Field descriptions

`physicalHigh`	See "PCI Encoded-Int Structure Constants" (page 5-19) for possible values.
`physicalMiddle`	To be provided later.
`physicalLow`	To be provided later.
`propAddress`	To be provided later.
`propLength`	To be provided later.

PCI Bus Range Property Structure

The PCi family defines the `PCIBusRangeProperty` structure to describe existing buses on a particular domain. This definition conforms to IEEE 1275-1994 Specification for PCI Bus Binding.

```
struct PCIBusRangeProperty {
    UInt32    lowBus;
    UInt32    highBus;
};
typedef PCIBusRangeProperty *PCIBusRangePropertyPtr;

typedef struct PCIBusRangeProperty PCIBusRangeProperty;
```

Field descriptions

`lowBus`	To be provided later.
`highBus`	To be provided later.

PCI Device Table Entry Header

PCI family defines `PCIDeviceTableEntryHeader` structure which is used by the PCI family for probing. This strucutre may also be passed in to a plug-in to separate cycles.

```
struct PCIDeviceTableEntryHeader {
    RegEntryRef       entry;
    char              name[32];
    UInt32            pciDomain;
    UInt32            pciBusNumber;
    UInt32            pciSecondBusNumber;
    UInt32            accessType;
    PCIRegPropertyPtr regProperty;
    ByteCount         regPropertyCount;
    IOAddress         ioBase;
    LogicalAddress    rangeBase;
};
typedef PCIDeviceTableEntryHeader *PCIDeviceTableEntryHeaderPtr;

typedef struct PCIDeviceTableEntryHeader PCIDeviceTableEntryHeader;
```

Field descriptions

`entry`	The device-specific identifier.
`name[32]`	The device name.
`pciDomain`	The domain to which the device belongs.
`pciBusNumber`	The bus number.
`pciSecondBusNumber`	In case of a bridge device, identifies the subordinate bus number.
`accessType`	Specifies whether the access is forwarded or not fowarded configuration cycles.
`regProperty`	Pointer to the “reg” property.
`regPropertyCount`	The number of “reg” properties associated with the device. A maximum of six properties can be supported.
`ioBase`	PCI base I/O address.
`rangeBase`	Device base I/O address.

Typedefs for Bridge Plugin Interface

The PCI provides type definitions that must be used for bridge plug-in prototyping.

```
typedef OSStatus (*InitializeFuncPtr)(void);

typedef void (*DefaultBridgeEnablerFuncPtr)(InterruptSetMember
setIDMember, void *refCon);

typedef InterruptSourceState
(*DefaultBridgeDisablerFuncPtr)(InterruptSetMember setIDMember, void
*refCon);

typedef InterruptMemberNumber
(*DefaultBridgeDispatcherFuncPtr)(InterruptSetMember setIDMember, void
*refCon, UInt32 theIntCount);

typedef OSStatus (*FinalizeFuncPtr)(void);
```

PCI Device Table Entry

The PCi family provides a subordinate PCI device description table entry structure definition in which the plug-in can store specific information.

```
struct PCIDeviceTableEntry {
    struct PCIDeviceTableEntry *nextDeviceEntry;
              PCIDeviceTableEntryHeader header;
              PCIBridgeDescriptorPtr BridgePlugin;
              UInt32 pluginSpecificStuff[16];
};
typedef PCIDeviceTableEntry *PCIDeviceTableEntryPtr;

typedef struct PCIDeviceTableEntry PCIDeviceTableEntry;
```

Field descriptions

`nextDeviceEntry` Pointer to the next device-specific identifier.

`header` The device table entry header used by the PCI family for probing.

`BridgePlugin` Pointer to the bridge plug-in.

`pluginSpecificStuff` Location where the plug-in can store specific information.

Typedefs for Plugin Interfaces

The PCI family provides type definitions that can be used for plug-in interfaces.

```
typedef OSStatus (*ConfigReadByteFuncPtr)(ConfigAddress configAddr,
UInt8 *value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*ConfigReadWordFuncPtr)(ConfigAddress configAddr,
UInt16 *value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*ConfigReadLongFuncPtr)(ConfigAddress configAddr,
UInt32 *value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*ConfigWriteByteFuncPtr)(ConfigAddress configAddr,
UInt8 value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*ConfigWriteWordFuncPtr)(ConfigAddress configAddr,
UInt16 value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*ConfigWriteLongFuncPtr)(ConfigAddress configAddr,
UInt32 value, PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOReadByteFuncPtr)(IOAddress ioAddr, UInt8 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOReadWordFuncPtr)(IOAddress ioAddr, UInt16 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOReadLongFuncPtr)(IOAddress ioAddr, UInt32 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOWriteByteFuncPtr)(IOAddress ioAddr, UInt8 value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOWriteWordFuncPtr)(IOAddress ioAddr, UInt16 value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IOWriteLongFuncPtr)(IOAddress ioAddr, UInt32 value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IntAckReadByteFuncPtr)(UInt8 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IntAckReadWordFuncPtr)(UInt16 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*IntAckReadLongFuncPtr)(UInt32 *value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*SpecialCycleWriteLongFuncPtr)(UInt32 value,
PCIDeviceTableEntryPtr pciDeviceHead);

typedef OSStatus (*InitDeviceEntryFuncPtr)(PCIDeviceTableEntryPtr
deviceDescriptor);

typedef OSStatus (*GetIOBaseFuncPtr)(PCIDeviceTableEntryPtr
deviceDescriptor, IOAddress *ioBase);
```

PCI Control Descriptor

The PCI family provides a template for the regular Dispatch table that can be used by plug-ins.

```
struct PCIControlDescriptor {
     PCIPluginHeader InterfaceHeader;
     PCIDeviceTableEntry PCIDeviceDescriptor;
     DriverDescription *TheDomainDriverDescription;
     InitializeFuncPtr InitializeFunc;
     ConfigReadByteFuncPtr ConfigReadByteFunc;
     ConfigReadWordFuncPtr ConfigReadWordFunc;
     ConfigReadLongFuncPtr ConfigReadLongFunc;
     ConfigWriteByteFuncPtr ConfigWriteByteFunc;
     ConfigWriteWordFuncPtr ConfigWriteWordFunc;
     ConfigWriteLongFuncPtr ConfigWriteLongFunc;
     IOReadByteFuncPtr IOReadByteFunc;
     IOReadWordFuncPtr IOReadWordFunc;
     IOReadLongFuncPtr IOReadLongFunc;
     IOWriteByteFuncPtr IOWriteByteFunc;
     IOWriteWordFuncPtr IOWriteWordFunc;
     IOWriteLongFuncPtr IOWriteLongFunc;
     IntAckReadByteFuncPtr IntAckReadByteFunc;
     IntAckReadWordFuncPtr IntAckReadWordFunc;
     IntAckReadLongFuncPtr IntAckReadLongFunc;
     SpecialCycleWriteLongFuncPtr SpecialCycleWriteLongFunc;
     InitDeviceEntryFuncPtr InitDeviceEntryFunc;
     GetIOBaseFuncPtr GetIOBaseFunc;
     FinalizeFuncPtr FinalizeFunc;
};
typedef struct PCIControlDescriptor PCIControlDescriptor;
typedef PCIControlDescriptor *PCIControlDescriptorPtr;
```

PCI Bridge Plugin Definitions

```
typedef void (*DefaultEnablerFuncPtr)(InterruptSetMember
setIDMember,void *refCon);

typedef InterruptSourceState
(*DefaultDisablerFuncPtr)(InterruptSetMember setIDMember, void *refCon);

typedef InterruptMemberNumber
(*DefaultDispatcherFuncPtr)(InterruptSetMember setIDMember, void *info,
UInt32 theIntCount);

typedef OSStatus (*InitializeFuncPtr)(RegEntryRef *entry);

typedef OSStatus (*FinalizeFuncPtr)(RegEntryRef *entry);
```

General Purpose PCI Masks

The PCI family provides enumerated values that can be used to check values in the configuration address.

```
enum {

kPCIconfigAddrReservedValue= 0x00000000,

kPCIconfigAddrReservedMask= 0xFF000000,

kPCIconfigAddrBusNumberMask= 0x00FF0000,

kPCIconfigAddrDeviceNumberMask = 0x0000F800,

kPCIconfigAddrFunctionNumberMask = 0x00000700,

kPCIconfigAddrRegisterNumberMask = 0x000000FC,

kPCIconfigAddrAccessTypeMask = 0x00000001,

kPCIregisterByteMask= 0x00000003,

kPCIregisterNotByteMask= 0xFFFFFFFC,

kPCIregisterWordMask= 0x00000002

};
```

PCI Encoded-Int Structure Constants

The PCI family provides enumerated values that describe the contents of the `physicalHigh` field in the PCI "reg" property structure.

```
enum {
kPCIPhysicalHighRelocatableMask = 0x80000000,
kPCIPhysicalHighRelocatable= 0x80000000,
kPCIPhysicalHighPrefetchableMask = 0x40000000,
kPCIPhysicalHighPrefetchable = 0x40000000,
kPCIPhysicalHighAliasedMask= 0x20000000,
kPCIPhysicalHighAliased= 0x20000000,
kPCIPhysicalHighSpaceCodeMask = 0x03000000,
kPCIPhysicalHighSpaceCodeConfig = 0x00000000,
kPCIPhysicalHighSpaceCodeIO= 0x01000000,
kPCIPhysicalHighSpaceCodeMemory = 0x02000000,
kPCIPhysicalHighSpaceCode64Bit = 0x03000000,
kPCIPhysicalHighBusMask= 0x00FF0000,
kPCIPhysicalHighDeviceMask= 0x0000F800,
kPCIPhysicalHighDevice0= 0x00000000,
kPCIPhysicalHighDevice1= 0x00000800,
kPCIPhysicalHighDevice2= 0x00001000,
kPCIPhysicalHighDevice3= 0x00001800,
kPCIPhysicalHighDevice4= 0x00002000,
kPCIPhysicalHighDevice5= 0x00002800,
kPCIPhysicalHighDevice6= 0x00003000,
kPCIPhysicalHighDevice7= 0x00003800,
kPCIPhysicalHighDevice8= 0x00004000,
kPCIPhysicalHighDevice9= 0x00004800,
kPCIPhysicalHighDevice10= 0x00005000,
kPCIPhysicalHighDevice11= 0x00005800,
kPCIPhysicalHighDevice12= 0x00006000,
kPCIPhysicalHighDevice13= 0x00006800,
kPCIPhysicalHighDevice14= 0x00007000,
kPCIPhysicalHighDevice15= 0x00007800,
```

```
kPCIPhysicalHighDevice16= 0x00008000,
kPCIPhysicalHighDevice17= 0x00008800,
kPCIPhysicalHighDevice18= 0x00009000,
kPCIPhysicalHighDevice19= 0x00009800,
kPCIPhysicalHighDevice20= 0x0000A000,
kPCIPhysicalHighDevice21= 0x0000A800,
kPCIPhysicalHighDevice22= 0x0000B000,
kPCIPhysicalHighDevice23= 0x0000B800,
kPCIPhysicalHighDevice24= 0x0000C000,
kPCIPhysicalHighDevice25= 0x0000C800,
kPCIPhysicalHighDevice26= 0x0000D000,
kPCIPhysicalHighDevice27= 0x0000D800,
kPCIPhysicalHighDevice28= 0x0000E000,
kPCIPhysicalHighDevice29= 0x0000E800,
kPCIPhysicalHighDevice30= 0x0000F000,
kPCIPhysicalHighDevice31= 0x0000F800,
kPCIPhysicalHighFunctionMask = 0x00000700,
kPCIPhysicalHighFunction0= 0x00000000,
kPCIPhysicalHighFunction1= 0x00000100,
kPCIPhysicalHighFunction2= 0x00000200,
kPCIPhysicalHighFunction3= 0x00000300,
kPCIPhysicalHighFunction4= 0x00000400,
kPCIPhysicalHighFunction5= 0x00000500,
kPCIPhysicalHighFunction6= 0x00000600,
kPCIPhysicalHighFunction7= 0x00000700,
kPCIPhysicalHighRegisterMask = 0x000000FF,
kPCIPhysicalHighRegisterVendorID = 0x00000000,
kPCIPhysicalHighRegisterDeviceID = 0x00000002,
kPCIPhysicalHighRegisterCommand = 0x00000004,
kPCIPhysicalHighRegisterRevisionID = 0x00000008,
kPCIPhysicalHighRegisterCacheLineSIze = 0x0000000C,
kPCIPhysicalHighRegisterHeaderType = 0x0000000E,
kPCIPhysicalHighRegisterBaseAddress = 0x00000010,
```

```
kPCIPhysicalHighRegisterBridgeBusInfo = 0x00000018,
kPCIPhysicalHighRegisterCardbusCIS = 0x00000028,
kPCIPhysicalHighRegisterSubsystemVendorID = 0x0000002C,
kPCIPhysicalHighRegisterExpansionROMBase = 0x00000030,
kPCIPhysicalHighRegisterInterruptLine = 0x0000003C
};
```

PCI Cycle AccessType

These access types are used for forwarding (0) or not fowarding (1) configuration cycles.

```
enum {
    kPCIaccessType0= 0,
    kPCIaccessType1= 1
};
```

Byte Swapping Routines

The Macintosh system firmware provides two routines that can be used to swap bytes between big-endian an little-endian data formats. The routines are as follows:

- EndianSwap16Bit
- EndianSwap32Bit

These routines are defined in the PCI.h file.

EndianSwap16Bit

Swaps bytes between big-endian and little-endian format.

```
extern pascal UInt16 EndianSwap16Bit (UInt16 data16)
```

`data16` 2-byte input

DISCUSSION

`EndianSwap16Bit` returns a byte swapped version of its input values and in this way, converts big-endian data to or from little-endian data.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space, user space, and interrupt level.

EndianSwap32Bit

Swaps bytes between big-endian and little-endian format.

```
extern pascal UInt32 EndianSwap32Bit (UInt32 data32)
```

`data32` 4-byte input

DISCUSSION

`EndianSwap32Bit` returns a byte swapped version of its input values and in this way, converts big-endian data to or from little-endian data.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space, user space, and interrupt level.

PCI Kernel Cycle Routines

The Some PCI cards may require PCI cycles other than ordinary memory access cycles. The Expansion Bus Manager provide routines that create other PCI cycle types such as the following:

The PCI Kernel routines provide the same functionality as the Expansion Bus Manager routines. These routines create PCI cycle types other than the ordinary memory access cycles. These cycle types are

- Configuration space cycles
- I/O space cycles
- Interrupt acknowledge cycles
- Special cycles

These routines are defined in the PCIKernel.h file.

PCIConfigReadByte

Reads a byte value at a specific address in PCI configuration space.

```
extern OSStatus PCIConfigReadByte(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt8 *value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`configAddr`	The configuration address offset. It can be a value between 0 and 255.
`value`	Pointer to the returned value.
function result	A result code. The result code `noerr` indicates that `PCIConfigReadByte` successfully read the byte value at the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node is not in the device tree.

DISCUSSION

The `PCIConfigReadByte` function reads the byte at the address in PCI configuration space for the device node specified by `entry`. The byte in PCI configuration space that is to be read is determined by the offset specified by `configAddr`. The byte value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIConfigReadWord

Reads a word value at a specific address in PCI configuration space.

```
extern OSStatus PCIConfigReadWord(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt16 *value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`configAddr`	The configuration address offset. It can be a value between 0 and 255.
`value`	Pointer to the returned 16-bit value as it would appear on the PCI bus. The function peforms the necessary byte swapping.
function result	A result code. The result code `noerr` indicates that `PCIConfigReadWord` successfully read the word value at the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node is not in the device tree.

DISCUSSION

The `PCIConfigReadWord` function reads the word at the address in PCI configuration space for the device node specified by `entry`. The word in PCI configuration space that is to be read is determined by the offset specified by `configAddr`. The word value that is read is returned in `*value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIConfigReadLong

Reads a long word value at a specific address in PCI configuration space.

```
extern OSStatus PCIConfigReadLong(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt32 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` Pointer to the returned 32-bit value as it would appear on the PCI bus. The function peforms the necessary byte swapping.

function result A result code. The result code `noerr` indicates that `PCIConfigReadLong` successfully read the long word value at the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node is not in the device tree.

DISCUSSION

The `PCIConfigReadLong` function reads the long word at the address in PCI configuration space for the device node specified by `entry`. The long word in PCI configuration space that is to be read is determined by the offset specified by `configAddr`. The long word value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIConfigWriteByte

Writes a byte to a specific address in PCI configuration space.

```
extern OSStatus PCIConfigWriteByte(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt8 value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`configAddr`	The configuration address offset. It can be a value between 0 and 255.
`value`	The 8-bit value.
function result	A result code. The result code `noerr` indicates that `PCIConfigWriteByte` successfully wrote the 8-bit value to the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIConfigWriteByte` function writes an 8-bit value to an address in PCI configuration space for the device node specified by `entry`. The address in PCI configuration space that is to be written to is determined by the offset specified by `configAddr`. The 8-bit value that is written is specified by `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIConfigWriteWord

Writes a word to a specific address in PCI configuration space.

```
extern OSStatus PCIConfigWriteWord(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt16 value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`configAddr`	The configuration address offset. It can be a value between 0 and 255.
`value`	The 16-bit value. The function performs the necessary byte swapping.
function result	A result code. The result code `noerr` indicates that `PCIConfigWriteWord` successfully wrote the 16-bit value to the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIConfigWriteWord` function writes a 16-bit value to an address in PCI configuration space for the device node specified by `entry`. The address in PCI configuration space that is to be written to is determined by the offset specified by `configAddr`. The 16-bit value that is written is specified by `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIConfigWriteLong

Writes a long word to a specific address in PCI configuration space.

```
extern OSStatus PCIConfigWriteLong(
                    RegEntryRef *entry,
                    PCIConfigAddress configAddr,
                    UInt32 value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`configAddr`	The configuration address offset. It can be a value between 0 and 255.
`value`	The 32-bit value. The function performs the necessary byte swapping.
function result	A result code. The result code `noerr` indicates that `PCIConfigWriteLong` successfully wrote the 32-bit value to the specified address in PCI configuration space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIConfigWriteLong` function writes a 32-bit value to an address in PCI configuration space for the device node specified by `entry`. The address in PCI configuration space that is to be written to is determined by the offset specified by `configAddr`. The 32-bit value that is written is specified by `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOReadByte

Reads a byte value at a specific address in PCI I/O space.

```
extern OSStatus PCIIOReadByte(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt8 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`ioAddr` The I/O address offset.

`value` Pointer to the returned 8-bit value.

function result A result code. The result code `noerr` indicates that `PCIIOReadByte` successfully read the 8-bit value at the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOReadByte` function reads an 8-bit value at an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be read is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 8-bit value that is read is returned in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOReadWord

Reads a word value at a specific address in PCI I/O space.

```
extern OSStatus PCIIOReadWord(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt16 *value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`ioAddr`	The I/O address offset.
`value`	Pointer to the returned 16-bit value.
function result	A result code. The result code `noerr` indicates that `PCIIOReadWord` successfully read the 16-bit value at the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOReadWord` function reads a 16-bit value at an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be read is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 16-bit value that is read is returned in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOReadLong

Reads a long word value at a specific address in PCI I/O space.

```
extern OSStatus PCIIOReadLong(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt32 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`ioAddr` The I/O address offset.

`value` Pointer to the returned 32-bit value.

function result A result code. The result code `noerr` indicates that `PCIIOReadLong` successfully read the 32-bit value at the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOReadLong` function reads a 32-bit value at an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be read is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 32-bit value that is read is returned in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOWriteByte

Writes a byte value to a specific address in PCI I/O space.

```
extern OSStatus PCIIOWriteByte(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt8 value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`ioAddr`	The I/O address offset.
`value`	The 8-bit value to be written.
function result	A result code. The result code `noerr` indicates that `PCIIOWriteByte` successfully wrote the 8-bit value to the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOWriteByte` function writes an 8-bit value to an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be written to is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 8-bit value that is to be written is specified in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOWriteWord

Writes a word value to a specific address in PCI I/O space.

```
extern OSStatus PCIIOWriteWord(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt16 value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`ioAddr` The I/O address offset.

`value` The 16-bit value to be written as it would appear on the PCI bus. THe function performs the necessary byte swapping.

function result A result code. The result code `noerr` indicates that `PCIIOWriteWord` successfully wrote the 16-bit value to the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOWriteWord` function writes a 16-bit value to an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be written to is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 16-bit value that is to written is specified in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIOWriteLong

Writes a long word value to a specific address in PCI I/O space.

```
extern OSStatus PCIIOWriteLong(
                    RegEntryRef *entry,
                    PCIIOAddress ioAddr,
                    UInt32 value);
```

`entry`	Pointer to an identifier that identifies a device node in the Name Registry.
`ioAddr`	The I/O address offset.
`value`	The 32-bit value to be written as it would appear on the PCI bus. The function performs the necessary byte swapping.
function result	A result code. The result code `noerr` indicates that `PCIIOWriteLong` successfully wrote the 32-bit value to the specified address in PCI I/O space. The result code `nrInvalidNodeErr` indicates that the specified device node identifier is not in the device tree.

DISCUSSION

The `PCIIOWriteLong` function writes a 32-bit value to an address in PCI I/O space for the device node pointed to by `entry`. The address in PCI I/O space that is to be written to is the sum of the assigned-addresses base address of the device and the offset to the I/O address specified by `ioAddr`. The 32-bit value that is to written is specified in `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIntAckReadByte

Reads a byte value that resulted from a PCI interrupt acknowledge cycle.

```
extern OSStatus PCIIntAckReadByte(
                    RegEntryRef *entry,
                    UInt8 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`value` Pointer to a buffer that will hold the 8-bit value read.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIntAckReadWord

Reads a word value that resulted from a PCI interrupt acknowledge cycle.

```
extern OSStatus PCIIntAckReadWord(
                    RegEntryRef *entry,
                    UInt16 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`value` Pointer to a buffer that will hold the 16-bit value read.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCIIntAckReadLong

Reads a long word value that resulted from a PCI interrupt acknowledge cycle.

```
extern OSStatus PCIIntAckReadLong(
                    RegEntryRef *entry,
                    UInt32 *value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`value` Pointer to a buffer that will hold the 32-bit value read.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCISpecialCycleWriteLong

Writes a long word value to the PCI bus that contains the specified device node.

```
extern OSStatus PCISpecialCycleWriteLong(
                    RegEntryRef *entry,
                    UInt32 value);
```

`entry` Pointer to an identifier that identifies a device node in the Name Registry.

`value` 32-bit value to be written.

DISCUSSION

Typically, computers running Mac OS 8 that implement a PCI bus do not use special cycles. Special cycle functionality is usually provided to maintain microprocessor cache coherency across the PCI bus.

Special cycles on the PCI bus are always long word write actions.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCISpecialCycleBroadcastLong

Writes a long word value to all the PCI buses in the system.

```
extern OSStatus PCISpecialCycleBroadcastLong (UInt32 value);
```

`value` The 32-bit value to be written.

DISCUSSION

Typically, computers running Mac OS 8 that implement a PCI bus do not use special cycles. Special cycle functionality is usually provided to maintain microprocessor cache coherency across the PCI bus.

Special cycles on the PCI bus are always long word write actions.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

PCI I/O Iterator Routines

These routines are used to use the `PCIIOIterator` data structure to get devices known to the PCI family.

These routines are defined in the PCIKernel. h file.

PCIGetDeviceData

Returns a list of descriptions for all devices known to the PCI family.

```
extern OSStatus PCIGetDeviceData(
                    ItemCount requestItemCount,
                    ItemCount *totalItemCountPtr,
                    PCIIOIteratorData *List);
```

`requestItemCount`
The number of device descriptions that can be entered into the buffer that was provided by the client.

`totalItemCountPtr`
The number of device descriptions found and placed into the buffer that was provided by the client. If the value is zero, no devices are known to the PCI family.

`List` Pointer to the buffer that was provided by the client. This buffer is to be filled with device description information.

function result A result code. The result code `noerr` indicates that `PCIGetDeviceData` successfully returned a list of device descriptions into the client provided buffer.

DISCUSSION

The `PCIGetDeviceData` function returns a list of descriptions of all devices known to the PCI family. The descriptions are placed in a client provided buffer pointed to by `List`. Each device description is contained in a `PCIOIteratorData` structure. For information about this data structure, see "PCIIOIteratorData Structure" (page 5-9).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space and user space.

PCINameGetDeviceData

Returns a list of descriptions for all devices with the specified name known to the PCI family .

```
extern OSStatus PCINameGetDeviceData(
                    char *Name,
                    ItemCount reqeustItemCount,
                    ItemCount *totalItemCountPtr,
                    PCIIOIteratorData *List);
```

`Name` A pointer to the C-string that represents the name of the device for which the PCI family is to be searched.

`requestItemCount`
The number of device descriptions that can be entered into the buffer that was provided by the client.

`totalItemCountPtr`
The number of device descriptions found and placed into the buffer that was provided by the client. If the value is zero, no devices of the specified name are known to the PCI family.

`List` Pointer to the buffer that was provided by the client. This buffer is to be filled with device description information meeting the name search criterion.

function result A result code. The result code `noerr` indicates that `PCINameGetDeviceData` successfully returned a list of device descriptions into the client provided buffer.

DISCUSSION

The `PCINameGetDeviceData` function returns a list of descriptions of all devices with the specified name known to the PCI family. The descriptions are placed in the client provided buffere pointed to by `List`. Each device description is contained in a `PCIOIteratorData` structure. For information about this data structure, see "`PCIIOIteratorData Structure` `(page 5-9)`."

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space and user space.

PCIDomainGetDeviceData

Returns a list of descriptions for all devices from the specified domain known to the PCI family .

```
extern OSStatus PCIDomainGetDeviceData(
                    UInt32 Domain,
                    ItemCount reqeustItemCount,
                    ItemCount *totalItemCountPtr,
                    PCIIOIteratorData *List);
```

`Domain` A value representing an electrically separate set of PCI busses.

`requestItemCount`
The number of device descriptions that can be entered into the buffer that was provided by the client.

`totalItemCountPtr`
The number of device descriptions found and placed into the buffer that was provided by the client. If the value is zero, no devices from the specified domain are known to the PCI family.

`List`	Pointer to the buffer that was provided by the client. This buffer is to be filled with device description information meeting the domain search criterion.
function result	A result code. The result code `noerr` indicates that `PCIDomainGetDeviceData` successfully returned a list of device descriptions into the client provided buffer.

DISCUSSION

The `PCIDomainGetDeviceData` function returns a list of descriptions of all devices from the specified domain known to the PCI family. The descriptions are placed in the client provided buffere pointed to by `List`. Each device description is contained in a `PCIOIteratorData` structure. For information about this data structure, see "`PCIIOIteratorData Structure`" `(page 5-9)`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space and user space.

PCIBusNumberGetDeviceData

Returns a list of descriptions for all devices from the specified PCI bus known to the PCI family .

```
extern OSStatus PCIBusNumberGetDeviceData(
                    UInt32 BusNumber,
                    ItemCount reqeustItemCount,
                    ItemCount *totalItemCountPtr,
                    PCIIOIteratorData *List);
```

`BusNumber` A value representing a unique PCI bus.

`requestItemCount`
The number of device descriptions that can be entered into the buffer that was provided by the client.

`totalItemCountPtr`
The number of device descriptions found and placed into the buffer that was provided by the client. If the value is zero, no devices from the specified PCI bus are known to the PCI family.

`List` Pointer to the buffer that was provided by the client. This buffer is to be filled with device description information meeting the bus number search criterion.

function result A result code. The result code `noerr` indicates that `PCIBusNumberGetDeviceData` successfully returned a list of device descriptions into the client provided buffer.

DISCUSSION

The `PCIBusNumberGetDeviceData` function returns a list of descriptions of all devices from the specified PCI bus known to the PCI family. The descriptions are placed in the client provided buffer pointed to by `List`. Each device description is contained in a `PCIOIteratorData` structure. For information about this data structure, see "`PCIIOIteratorData Structure` `(page 5-9)`."

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space and user space.

PCIConfigAddressGetDeviceData

Returns a description of the device at the specified PCI configuration address.

```
extern OSStatus PCIConfigAddressGetDeviceData(
                    PCIConfigAddress ConfigAddress,
                    ItemCount requestItemCount,
                    ItemCount *totalItemCountPtr,
                    PCIIOIteratorData *List);
```

`ConfigAddress`
A value representing a unique PCI configuration address. This address must be made up of the bus number, device function number, and register number as defined in the `PCIAssigedAddress` structure. For information about this data structure, see "PCIIOIteratorData Structure" (page 5-9)

`requestItemCount`
The number of device descriptions that can be entered into the buffer that was provided by the client.

`totalItemCountPtr`
The number of device descriptions found and placed into the buffer that was provided by the client. This value should never be greater than one. If the value is zero, no devices at the specified address are known to the PCI family.

`List` Pointer to the buffer that was provided by the client. This buffer is to be filled with device description information meeting the configuration address search criterion.

function result A result code. The result code `noerr` indicates that `PCIConfigAddressGetDeviceData` successfully returned a device description into the client provided buffer.

DISCUSSION

The `PCIConfigAddressGetDeviceData` function returns a description of the devices at the specified PCI configuration address known to the PCI family. The description is placed in the client provided buffer pointed to by `List`. The device description is contained in a `PCIOIteratorData` structure. For information about this data structure, see "`PCIIOIteratorData Structure` `(page 5-9)`."

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space and user space.

PCI Plugin Interface Routines

The PCI plugin interface routines can be used to interface between a PCI plugin and the PCI family expert.

PCIPluginInitialize

Initializes the PCI host bridge device.

```
extern OSStatus PCIPluginInitialize (RegEntryRef *entry);
```

function result A result code. The result code `noerr` indicates that `PCIPluginInitialize` successfully initialized the host bridge device.

DISCUSSION

This function is always the first function called within this plugin. If this function is called and the PCI bridge device has been initialized already, the function may be empty. If however, the PCI device has not been initiliazed by this function, be sure to clear any prior state of the PCI bridge device. In addition, upon completion of the call, the PCI bridge device must be in a known state of device operation.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space, and interrupt level, but must not block.

PCIPluginConfigReadByte

Generates a configuration read byte cycle addressed to a device.

```
extern OSStatus PCIPluginConfigReadByte(
                    ConfigAddress configAddr,
                    UInt8 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the read value.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigReadByte` successfully read the byte value at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigReadByte` function generates a read byte cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The byte value that is read is returned in `*value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginConfigReadWord

Generates a configuration read word cycle addressed to a device.

```
extern OSStatus PCIPluginConfigReadWord(
                    ConfigAddress configAddr,
                    UInt16 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the read value.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigReadWord` successfully read the word value at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigReadWord` function generates a read word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The word value that is read is returned in `*value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginConfigReadLong

Generates a configuration read long word cycle addressed to a device.

```
extern OSStatus PCIPluginConfigReadLong(
                    ConfigAddress configAddr,
                    UInt32 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the read value.

`pciDeviceEntry` Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigReadLong` successfully read the long word value at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigReadLong` function generates a read long word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The long word value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginConfigWriteByte

Generates a configuration write byte cycle addressed to a device.

```
extern OSStatus PCIPluginConfigWriteByte(
                    ConfigAddress configAddr,
                    UInt8 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` The value to be written.

`pciDeviceEntry` Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigWriteByte` successfully writes the byte value at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigWriteByte` function generates a write byte cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The byte value that is written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginConfigWriteWord

Generates a configuration write word cycle addressed to a device.

```
extern OSStatus PCIPluginConfigWriteWord(
                    ConfigAddress configAddr,
                    UInt16 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` The word value to be written.

`pciDeviceEntry` Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigWriteWord` successfully writes the word value at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigWriteWord` function generates a write word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The word value that is written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginConfigWriteLong

Generates a configuration write long word cycle addressed to a device.

```
extern OSStatus PCIPluginConfigWriteLong(
                    ConfigAddress configAddr,
                    UInt32 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`configAddr` The configuration address offset. It can be a value between 0 and 255.

`value` The 32-bit value to be written.

`pciDeviceEntry` Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginConfigWriteLong` successfully writes the long word value found at the specified address in the device configuration space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginConfigWriteLong` function generates a write long word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `configAddr` is an offset into the device configuration space. The long word value that is written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOReadByte

Generates an I/O read byte cycle addressed to a device.

```
extern OSStatus PCIPluginIOReadByte(
                    IOAddress ioAddr,
                    UInt8 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the read value.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOReadByte` successfully read the byte value at the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOReadByte` function generates a I/O read byte cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The byte value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOReadWord

Generates an I/O read word cycle addressed to a device.

```
extern OSStatus PCIPluginIOReadWord(
                    IOAddress ioAddr,
                    UInt16 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the 16-bit read value.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOReadWord` successfully read the word value at the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOReadWord` function generates a I/O read word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The word value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOReadLong

Generates an I/O read long word cycle addressed to a device.

```
extern OSStatus PCIPluginIOReadLong(
                    IOAddress ioAddr,
                    UInt32 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` Pointer to the location in which to return the 32-bit read value.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOReadLong` successfully read the long word value at the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOReadLong` function generates a I/O read long word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The long word value that is read is returned in *`value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOWriteByte

Generates an I/O write byte cycle addressed to a device.

```
extern OSStatus PCIPluginIOWriteByte(
                    IOAddress ioAddr,
                    UInt8 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` The byte value to be written.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOWriteByte` successfully wrote the byte value to the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOWriteByte` function generates a I/O write byte cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The byte value that is to be written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOWriteWord

Generates an I/O write word cycle addressed to a device.

```
extern OSStatusPCIPluginIOWriteWord(
                    IOAddress ioAddr,
                    UInt16 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` The 16-bit value to be written.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOWriteWord` successfully wrote the word value to the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOWriteWord` function generates a I/O write word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The word value that is to be written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIOWriteLong

Generates an I/O write long word cycle addressed to a device.

```
extern OSStatus PCIPluginIOWriteLong(
                    IOAddress ioAddr,
                    UInt32 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`ioAddr` The I/O address offset. It can be a value between 0 and 255.

`value` The 32-bit value to be written.

`pciDeviceEntry`
Pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginIOWriteLong` successfully wrote the long word value to the specified address in the device I/O space. The result code `nrInvalidNodeErr` indicates that the specified device was not found.

DISCUSSION

The `PCIPluginIOWritelong` function generates an I/O write word cycle for the device described by `pciDeviceEntry`. For more information about this data structure, see "PCI Device Table Entry" (page 5-15). The `IOAddr` is an offset into the device I/O space. The long word value that is to be written is `value`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIntAckReadByte

Generates a PCI interrupt acknowledge read byte cycle addressed to the specified device domain.

```
extern OSStatus PCIPluginIntAckReadByte(
                    UInt8 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`value` Pointer to alocation at which to return the 8-bit value.

`pciDeviceEntry`
Pointer to a PCI specific device description data structure.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt leve, but must not block.

PCIPluginIntAckReadWord

Generates a PCI interrupt acknowledge read word cycle addressed to the specified device domain.

```
extern OSStatus PCIPluginIntAckReadWord(
                    UInt16 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`value` Pointer to alocation at which to return the 16-bit value.

`pciDeviceEntry`
Pointer to a PCI specific device description data structure.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginIntAckReadLong

Generates a PCI interrupt acknowledge read long word cycle addressed to the specified device domain.

```
extern OSStatus PCIPluginIntAckReadLong(
                    UInt32 *value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`value` Pointer to a location at which to return the 32-bit value.

`pciDeviceEntry`
Pointer to a PCI specific device description data structure.

DISCUSSION

Mac OS 8 does not use PCI interrupt acknowledge cycles. This functionality is provided to allow a driver to create a cycle for a PCI device, such as an 8259-style interrupt controller, which requires such a cycle.

Interrupt acknowledge cylces for PCI are always read actions. The target device node must be a single node capable of responding to interrupt acknowledge cycles.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginSpecialCycleWriteLong

Generates a special cycle write long word value to the device domain.

```
extern OSStatus PCIPluginSpecialCycleWriteLong(
                    UInt32 value,
                    PCIDeviceTableEntryPtr pciDeviceEntry);
```

`value` The 32-bit value that is to be written onto the PCI bus.

`pciDeviceEntry`
Pointer to a PCI specific device description data structure.

DISCUSSION

Typically, computers running Mac OS 8 that implement a PCI bus do not use special cycles. Special cycle functionality is usually provided to maintain microprocessor cache coherency across the PCI bus.

Special cycles on the PCI bus are always long word write actions.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginInitDeviceEntry

Initializes a PCI specific device description data structure.

```
extern OSStatus PCIPluginInitDeviceEntry (PCIDeviceTableEntryPtr
                    deviceDescriptor);
```

`deviceDescriptor`
A pointer to the PCI specific device description data structure.

function result A result code. The result code `noerr` indicates that `PCIPluginInitDeviceEntry` successfully initialized the device description data structure.

DISCUSSION

This function is always the first function called on the device description data structure with this same device descriptor.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space or itnerrupt level, but must not block.

PCIPluginGetIOBase

Returns the I/O space base address for the device domain.

```
extern OSStatus PCIPluginGetIOBase(
                    PCIDeviceTableEntryPtr deviceDescriptor,
                    IOAddress *ioBase);
```

`deviceDescriptor`
Pointer to the PCI specific device description data structure.

`ioBase` Pointer to the location where the I/O base address of this domain is to be returned.

function result A result code. The result code `noerr` indicates that `PCIPluginGetIOBase` successfully retrieves the I/O base address for the device domain. .

DISCUSSION

This function communicates to the PCI family where I/O space begins.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCIPluginFinalize

Disables the PCI host bridge device.

```
extern OSStatus PCIPluginFinalize (void);
```

function result A result code. The result code `noerr` indicates that `PCIPluginFinalize` successfully disabled the host bridge device.

DISCUSSION

Upon completion, this function must disable the PCI host bridge device. If the PCI host bridge device does not need to be disabled, that is, it has already been disabled, this function may be empty. If however, the PCI device has not been disabled by this function, the device must not assume the state in which the device was left prior to the calling of this function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level, but must not block.

PCI Bridge Plug-in Routines

The PCI bridge plug-in routines can be used by a bridge plug-in to interface to the PCI expert. The bridge plugin provides functionality for PCI-to-PCI configurations that use for PCI bridge devices.

PCIBridgePluginInitialize

Initializes PCI bridge devices.

```
extern OSStatus PCIBridgePluginInitialize (RegEntryRef * entry);
```

function result A result code. The result code `noerr` indicates that `PCIBridgePluginGetInitialize` successfully enabled the PCI bridge devices.

DISCUSSION

This function enables PCI bridge devices which allow comunication across a PCI bridge.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space only.

DefaultBridgeEnabler

Invokes the bridge interrupt enabler routine.

```
extern void DefaultBridgeEnabler(
                    InterruptSetMember setIDMember,
                    void *refCon);
```

`setIDMember` Member set ID of the interrupt source tree (IST) member requesting service. The IST member is retrieved from the Name Registry..

`refCon` 32-bit reference constant registered with the IST member.

DISCUSSION

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

DefaultBridgeDisabler

Invokes the bridge disabler routine.

```
extern InterruptSourceState DefaultBridgeDisabler(
                    InterruptSetMember setIDMember,
                    void *refCon);
```

`setIDMember` Member set ID of the interrupt source tree (IST) member requesting service. The IST member is retrieved from the Name Registry.

`refCon` 32-bit reference constant registered with the IST member.

DISCUSSION

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and interrupt level.

DefaultBridgeDispatcher

Invokes transversal interrupt service routine (ISR).

```
extern InterruptMemberNumber DefaultBridgeDispatcher(
                    InterruptSetMember setIDMember,
                    void *refCon,
                    UInt32 theIntCount);
```

`setIDMember`	Member set ID of the IST member requesting service.
`refCon`	32-bit reference constant registered with the IST member.
`theIntCount`	Count of the number of interrupts processed, including the current one.

DISCUSSION

This function does not handle the actual interrupt but routes interrupts that are to be serviced.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	Yes

CALLING RESTRICTIONS

Can be called from kernel space and hardware interrupt level.

PCIBridgePluginFinalize

Shuts down PCI bridge devices.

```
extern OSStatus PCIBridgePluginFinalize (RegEntryRef * entry);
```

function result A result code. The result code `noerr` indicates that `PCIBridgePluginFinalize` successfully shut down the PCI bridge devices.

DISCUSSION

This function shuts down PCI bridge devices and disables communication across a PCI bridge.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Can be called from kernel space only.

About the Nubus Family

Contents

About the Nubus Family

This chapter introduces the NuBus family. It describes the NuBus family architecture and it software components. This chapter is intended for developers who will be writing NuBus plug-ins or applications that control NuBus devices.

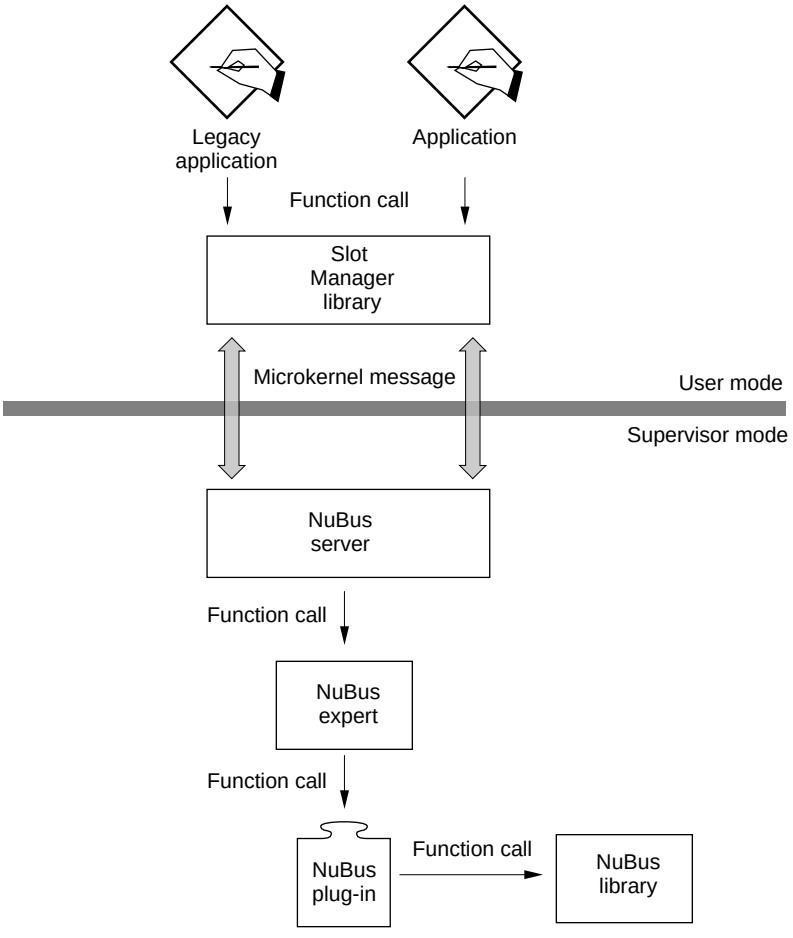
The NuBus family provides driver management services for NuBus devices running in the Mac OS 8 environment. The NuBus family is a low-level family that conforms to the I/O architecture requirements by providing the following software components:

- NuBus expert
- NuBus server
- NuBus Library (Not available in this release)
- Slot Manager Application Library, called Slots (Not available in this release)

The NuBus family implements the single-task activation model. In this activation model, the NuBus family receives requests placed into queues by the NuBus library or Slot Manager library in user space and receives interrupts from the plug-ins in kernel space. For more information about the single-task activation model, see “About the I/O Architecture” (page 1-3).

Figure 6-1 shows the Nubus family architecture.

Figure 6-1 Nubus Family Software Diagram



NuBus Expert

The NuBus expert manages physical NuBus devices within Mac OS 8 on a particular bus. The NuBus expert conforms to the Macintosh native driver model which defines a packaging format for generic and family drivers. The driver package for NuBus is a CFM code fragment that resides in Macintosh ROM.

NuBus is a static bus. As a result, the NuBus expert characteristics are established at start-up and do not change until the next time the system is started.

Once the NuBus expert successfully initializes or updates the NuBus environment, the NuBus expert terminates. However, while the NuBus expert exists, it is does the following:

- Discovers NuBus cards at start-up
- Establishes logical addresses for standard slot and super slot addresses
- Initializes its interrupt structure
- Advertises device information to NuBus drivers through Name Registry properties
- Advertises NuBus devices to High-Level families through Name Registry RegEntryRefs entries
- Matches file-system based NuBus drivers to NuBus devices

Discovering NuBus Cards

The NuBus expert attempts to discover all installed cards. The NuBus expert checks whether a Slot ROM exists at a particular slot base address. If a slot ROM is discovered, the NuBus checks for the following two slot resources:

- `BoardId`
- `sRsrcType`

If the NuBus expert cannot locate both of these resources for a card, the NuBus expert ignores the card because the Mac OS 8 environment will not recognize the card. The NuBus expert logs a message, indicating an incompatible card, to the System Logging Manager.

Establishing Logical Addresses

After the NuBus expert discovers a card, it negotiates address mappings for its slots, the NuBus expert discovers NuBus cards. The NuBus expert establishes logical addresses for standard slot and super slot addresses. The NuBus expert allocates address mapping for its particular bus implementation. The NuBus expert then requests a logical to physical address mapping from the memory allocator for each slot space address range. Every NuBus card found is allocated 16 MB of logical addresses for standard slot space and 256 MB for super slot space. These assignments can be found in the "AAPL,address" property of the card's `RegEntryRef` entry.

If a mapping request fails for either a standard slot or super slot, a fatal hardware configuration error is logged to the System Logging Manager.

Logical address assignments for NuBus may be different from its physical address assignments. Do not attempt to construct a logical address by using a card's slot number. Always obtain your card's logical address using the "AAPL,address" property.

Initializing Its Interrupt Structure

Once the NuBus expert negotiates address mappings for it slots, the NuBus expert creates an interrupt set by calling the `CreateInterruptSet` function. By creating an interrupt set, the NuBus family can handle a bus hardware interrupt structure specific to its implementation.

During the probe process, that is, while the NuBus expert uses its IOIterator to discover all NuBus cards, the NuBus expert creates an `InterruptSetMember` element for each NuBus card discovered. After all cards are discovered, the NuBus expert installs its interrupt handler into the `InterruptSetMember` element passed in the "driver-ist" property. This master interrupt handler is represented by the NuBus InterruptSet.

Advertising Device Information to NuBus Drivers

If the NuBus expert locates a card with the `BoardId` and `sRsrcType` resource, the NuBus expert creates a `RegEntryRef` entry for the card within the device tree in the Name Registry.

Note
The NuBus expert creates the `RegEntryRef` entry as a child of the `RegEntryRef` entry it was passed.

The NuBus expert attaches specific device information properties to the `RegEntryRef` entry before making it available to the system. The specific device information properties that describe the device are as follows:

- "assigned-addresses"
- "reg"
- "name"
- "AAPL,address"
- "AAPL,slot"
- "driver-ist"
- "driver-description"

Once the NuBus expert constructs the `RegEntryRef` entry and attaches the device properties, it installs the `RegEntryRef` entry into the device tree of the Name Registry. Once installed, the NuBus family advertises the existence of the card to other families.

"assigned-addresses" Property

The "assigned-addresses" property contains the physical slot address assignment for a NuBus card. The physical address might not be the same as the logical address.

“reg” Property

The “reg” property contains the physical slot address assignments which are identical to the “assigned-addresses” property.

“name” Property

The “name” property is used to match NuBus devices to the appropriate plug-in. The value of the “name” property is derived from the Nubus board ID. This board ID is the Slot ROM board ID value that is one of two minimum required slot resources that must be present in order for the NuBus expert to consider the NuBus card compatible.

The format for this property is:

“AAPL,nubus-boardXXXX”

where XXXX is your card’s BoardId value represented as 4-digit hexadecimal ASCII that is, leading zeroes and lowercase hexadecimal digits.

“AAPL,address” Property

The “AAPL, address” property contains the logical slot address assignment for a NuBus card. The logical address might not be the same as the physical address. You should use the “assigned-addresses” property to obtain the physical address for a NuBus card.

“AAPL,slot” Property

The “AAPL, slot” property contains the physical slot number into which the NuBus card has been installed. The valid value is in the range 0x9 through 0xE. This value can be used to compute the physical slot address but not the logical slot assignment.

“driver-ist” Property

The “driver-ist” property contains the Interrupt Source Tree (IST) member and set value used to install interrupts. This property provides the NuBus driver an `InterruptSetMember` structure used to install the card’s interrupt handler.

“driver-description” Property

The “driver-description” property contains the driver description structure. This property is used to match drivers with device, set-up and maintain a driver’s run-time environment, and declare a driver’s supported services. The structure contains the `nameInfoStr` element. This element is used to match the name property in the Name Registry.

Advertising NuBus Devices to High-Level Families

The NuBus expert also advertises the NuBus devices it has found to high-level families. The NuBus expert does this by placing a `RegEntryRef` entry in the Name Registry device tree for each card it has found. The high-level families, such as the Open Transport family, can then access these NuBus devices.

NuBus Server

The NuBus server services client requests for access to slot ROM information. The NuBus server does this on behalf of applications via the Slot Manager application Library (called Slots) and on behalf of plug-ins via NuBus Library.

NuBus Plug-in

Not available in this release.

NuBus Library

Not available in this release.

Slot Manager Library

The Slot Manager library is provided for compatibility reasons only. These functions will be called by applications. Because the I/O architecture provides address protection for tasks running in kernel space, the Slot Manager library no longer supports some functions previously supported in the Slot Manager. The Slot Manager functions are only a subset of the ones Slot Manager provides for earlier releases.

These routines are defined in the Slots.h file.

Note
This is not currently implemented.

Block Storage Family Reference

Contents

This chapter briefly describes basic block storage family concepts and provides a complete reference to its client and plug-in programming interfaces. The block storage family is the part of Mac OS 8 that abstracts the characteristics of, and operations on, large-capacity random-access physical storage devices such as hard disk drives, CR-ROM drives, floppy disk drives, and so forth. Its interface frees clients from needing specific information about individual devices. The block storage family is new in Mac OS 8—it has no direct counterpart in previous versions of Mac system software.

The block storage family mediates between clients seeking to access or configure random-access physical storage devices on the one hand, and the block storage plug-ins and other I/O families that actually control the devices on the other. The block storage family itself does not directly configure or access physical devices—it translates the client requests into calls to the appropriate block storage plug-in.

The chief clients of the block storage family are the file systems family and its plug-ins, which read and write data in files located on storage devices such as hard disks. Other clients include applications such as disk utility programs that configure storage devices. Any software—be it an application, another I/O family, or a server program—that requests a service from the block storage family is by definition a client.

If you develop client software, you need to understand the programming interfaces provided by the block storage family for its clients, described in "Block Storage Client Constants and Data Types" (page 7-27) and "Block Storage Client Functions" (page 7-77).

Block storage plug-ins are the software modules that actually provide the services requested by clients. Some block storage plug-ins know how to manipulate a given physical device or set of devices. Device drivers for hard disks connected via a SCSI bus are one example of this type of block storage plug-in—they send read and write commands to a disk; drivers for RAID devices are another. Other block storage plug-ins understand partitioning formats and carry out requests to format a block storage device in specific ways.

Apple provides a base set of block storage plug-ins. You can develop additional plug-ins that extend the set of devices and partitioning formats that can be supported through the block storage family.

If you want to develop a block storage plug-in, you need to understand the programming interfaces provided by the block storage family for its plug-ins, described in "Block Storage Plug-in Constants and Data Types" (page 7-49) and

"Block Storage Plug-in Functions" (page 7-157). Typically, you also need to know the client programming interface of another I/O family, such as SCSI or ATA, because the device that your plug-in manages is connected to the system by a particular bus technology. You can find information about different I/O families in other chapters in this book.

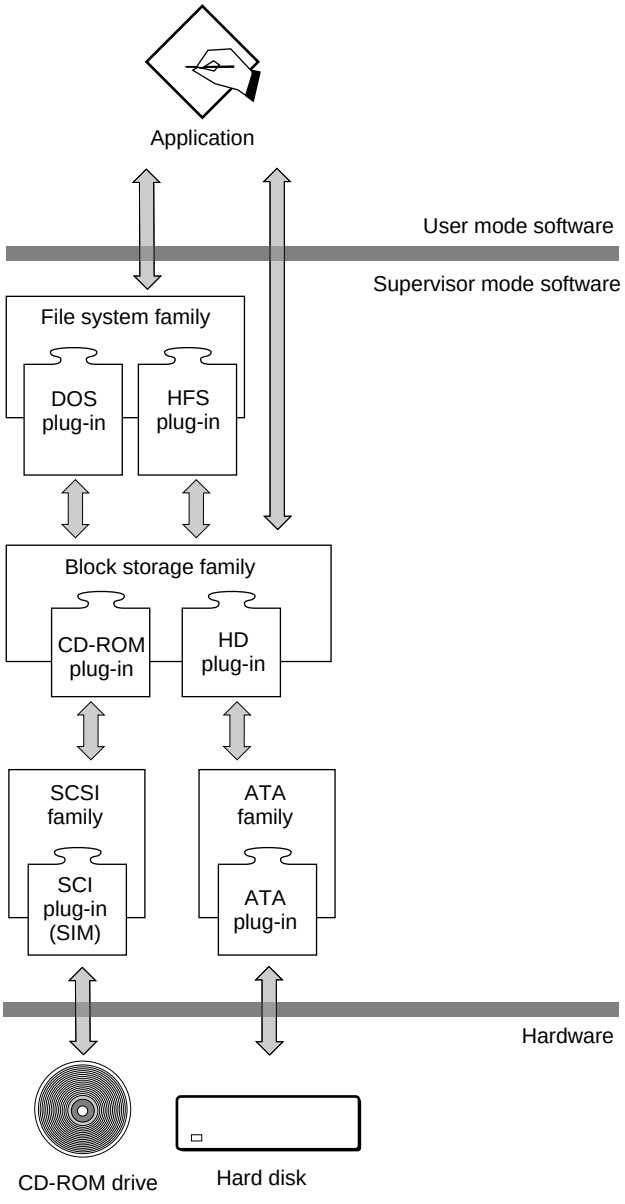
This chapter assumes that you are familiar with the architecture of the Mac OS 8 microkernel and the I/O subsystem. For information on the Mac OS 8 microkernel, see *Microkernel and Core System Services*. For information on the I/O architecture, see "About the I/O Architecture" (page 1-3).

About The Block Storage Family

The block storage family supports distributed storage system configuration in Mac OS 8—a given device contains all information about how it or its media fits into the storage system. This approach enables "plug and play" as well as "unplug and play"—that is, when attaching or removing a device, a user need not modify a system configuration file.

Most applications won't communicate directly with the block storage family; instead they call the file systems family to read and write data in files stored on random-access storage devices. Specialized applications like disk configuration utilities, however, do call the block storage family directly. Figure 7-1 shows the relationship of an application to the block storage family, other system software components, and devices accessible through the block storage family.

Figure 7-1 Relationship of block storage family to other software



To understand a little more of the relationship between an application, the block storage family, other software components of the I/O subsystem, and system hardware, consider the following example and refer to Figure 7-1. An application calls a File Manager function to read data from a file. The file is stored on a CD-ROM disc formatted as an HFS volume. The disc is loaded in a CD-ROM drive attached to the computer by a SCSI bus.

The File Manager forwards the application's read request to the file systems family. The file systems family calls one of its plug-ins—the one that understands the HFS volume format.

The selected file systems plug-in calls a block storage function to read the data. The block storage family forwards the request to one of its plug-ins—the one that understands how to read data from the CD-ROM disc on which the file is stored. The block storage plug-in is the device driver for the CD-ROM drive.

The block storage plug-in calls a SCSI family function to read the data. The SCSI family forwards the request to one of its plug-ins—the one that controls the SCSI bus to which the CD-ROM drive is connected. The SCSI plug-in is a SCSI Interface Module (SIM). It carries the read request to the CD-ROM drive.

Once the data on the CD-ROM disc is accessed, it is returned to the application along a reverse path through the families and plug-ins just described. As you can see, the path of an I/O request can travel through several software components. Each component offers well-defined services through its programming interfaces.

To make good use of the programming interfaces provided by the block storage family, you should understand the following:

- stores
- partitions
- containers
- connection-based services

Plug-in developers also need to know about the different types of plug-ins defined by the block storage family, how plug-ins are discovered by Mac OS 8, and the block storage activation model.

All of these topics are introduced the sections that follow.

Stores

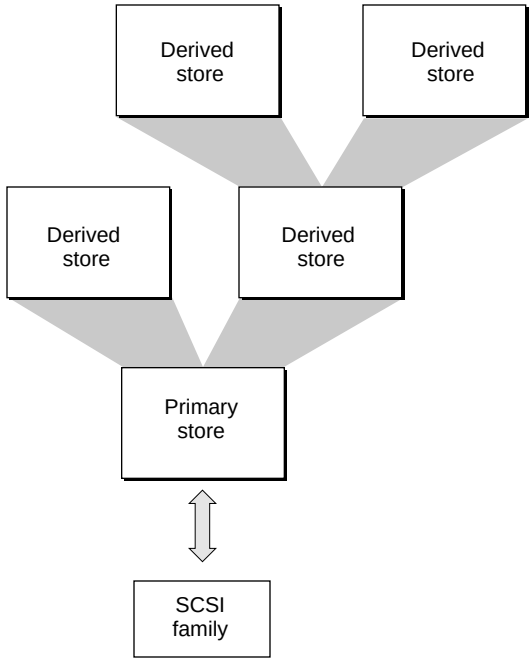
A **store** is an abstraction for a physical storage device. A store can represent up to 2^{64} linearly addressable bytes of data, starting from address 0, that can be randomly accessed on a physical device. Each store has its own read and write block sizes. For a given store, the read and write block sizes can be different.

The addresses belonging to a store (0 to 2^{64} or some part thereof) correspond to addresses valid for a given physical device or another store. The relationship between two stores or between a store and a physical device is called a **mapping**—it defines what byte of data a store returns when it is asked for a specific byte. The process of translating addresses between two stores or between a store and a physical device is also called a mapping (or sometimes a mapping operation).

A **primary store** maps to a physical device, such as a hard disk, that is beyond the awareness and control of the block storage family. A **derived store** maps to a primary store or another derived store. A store from which no other stores are derived is referred to as a **leaf store** or a *terminal store*.

Derived stores can be nested. Figure 7-2 illustrates the relationship between a primary store and two levels of derived stores. The block storage family supports several levels of stores. A series of nested derived stores ends in a primary store that maps to a physical device on which the data is actually stored.

Figure 7-2 Primary and derived stores



The block storage family automatically creates and maintains the store hierarchy. You can programtically extend and modify the hierarchy.

The store hierarchy consists of nodes connected together in a general graph structure, starting with a origin point called the **root**. All nodes in the hierarchy can be described by a path through the hierarchy starting from the root.

Nodes directly connected to the root are children of the root. A child node can in turn can have its own children, in which case it is also simultaneously a parent node. Nodes that have no children are leaf nodes.

Each node in the store hierarchy, with the exception of the root, represents a specific store. All nodes representing primary stores are children of the root. A node representing a derived store is always a child node. It can also be a parent node if other derived stores are created from it.

Each node has a set of properties, consisting of a property name and a property value, that provide information about the store.

A node is a member of a set. Members of a given set exist at the same level in the hierarchy and share a relationship to another designated node or nodes. The children of a given node constitute a set of sibling nodes. Unlike a strict tree structure, in the store hierarchy a child node can have more than one parent node. As a result, the set that includes the parent nodes for a given child can have more than one member.

Further discussion in this chapter refers to parent stores and child stores. The terms *parent store* and *child store* are relative to a given I/O request that flows from one store to another. The terms are a shorthand way of referring to the work performed by the mapping plug-ins for the two stores in processing the I/O request.

A **child store** is the store whose mapping plug-in performs the address translation for an I/O request and forwards the request to another store. The mapping plug-in translates addresses that are valid in the store it manages (the child store) into addresses that are valid in the store to which the mapping plug-in forwards the I/O request (the parent store).

A **parent store** is the store whose mapping plug-in receives the forwarded I/O request.

The condition of being a child store or a parent store is transient. If an I/O request flows through several layers of stores, a store can be characterized as

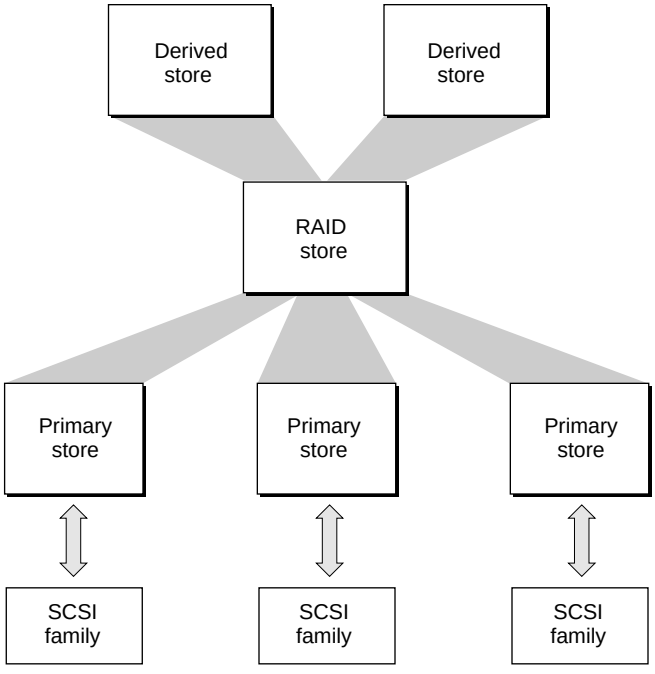
- a parent store when its mapping plug-in receives the request
- a child store when its mapping plug-in translates the addresses in the I/O and forwards the request to the next store

Because it maps to a physical device rather than another store, a primary store cannot be a child store.

(Don't confuse *parent store* and *child store* with the terms *parent node* and *child node*, which refer to the relationhip between given nodes in the store hierarchy.)

If a derived store could have only a 1-to-1 mapping of bytes to its parent store, derived stores would not be very interesting. However, a store can be mapped to a portion of another store. Furthermore, multiple stores can be aggregated together and mapped into a single store, allowing for software implementation of devices like RAID. Figure 7-3 shows a store hierarchy resulting from a RAID device. The two derived stores at the top of the figure each correspond to a portion of the RAID store. The three primary stores at the bottom of the figure are aggregated to form a single store—the RAID store.

Figure 7-3 RAID store hierarchy



Partitions

A physical device has one or more partitions. A **partition** is a portion of a device that can be treated as if it were a separate and distinct physical device—in other words, a virtual device. Partitions are typically allocated for an operating system, a file system, or a device driver. A partition map describes how the device is split up (partitioned) into virtual devices.

In the block storage world, each store has one or more partitions. Because the block storage family allows stores to be both subdivided and aggregated, the relationship between stores is referred to as a mapping, but the place where the mapping information is persistently stored is still referred to as a **partition map**.

Not all stores have partition maps. Only stores from which other stores are derived contain a partition map. You can think of an entry in a partition map as

a blueprint for a new derived store. Leaf stores consist of a single partition and do not contain a partition map.

Most operating systems, in creating virtual devices corresponding to the partitions described in a partition map, allow only a two-level hierarchy of physical devices and virtual devices. The block storage family allows any store to contain a partition map, resulting in a multilevel hierarchy of stores.

A given partitioning format is usually closely tied to a specific operating system—few support multiple partitioning formats. The block storage family eliminates this restriction. Using partitioning plug-ins, it can accomodate multiple partitioning formats, each treated equally.

Containers

•••To be provided•••

Connections

A connection is the mechanism by which the block storage family provides access to a store or a container. All operations that modify a store or a container require a connection. You get a connection ID when you open a store or a container.

You can open a connection for exclusive access. In that case, the block storage family does not allow another connection until you close the exclusive connection.

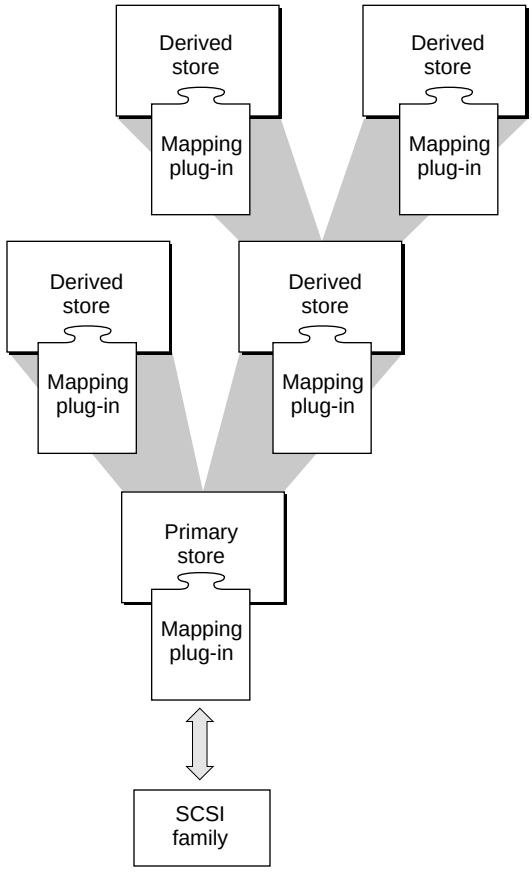
Plug-ins

The block storage family defines 3 types of plug-ins to provide services to block storage clients: mapping plug-ins, partitioning plug-ins, and container plug-ins.

Mapping Plug-ins

Each store has one **mapping plug-in**, a sofware module that implements the mapping between the store and its parent store or physical device. Figure 7-4 shows mapping plug-ins attached to stores.

Figure 7-4 Mapping plug-ins



Mapping plug-ins, on getting an I/O request, translate the store addresses in the I/O request into the corresponding addresses in the parent store or the device.

A primary store mapping plug-in is typically more complex than a derived store mapping plug-in. A primary store mapping plug-in is actually the device driver for the physical device. It knows how to translate addresses in an I/O request into operations on the physical device. For example, a SCSI hard-disk mapping plug-in knows how to create a SCSI Command Descriptor Block and

pass it in a function call to the SCSI family. A primary store mapping plug-in is responsible for all interactions with other I/O families (if any).

Apple provides a derived store mapping plug-in that performs the address translation needed in passing an I/O request to a parent store. You can provide additional derived store mapping plug-ins to implement more elaborate storage systems, such as RAID.

All mapping plug-ins have the same interface, described in "Block Storage Plug-in Constants and Data Types" (page 7-49) and "Block Storage Plug-in Functions" (page 7-157).

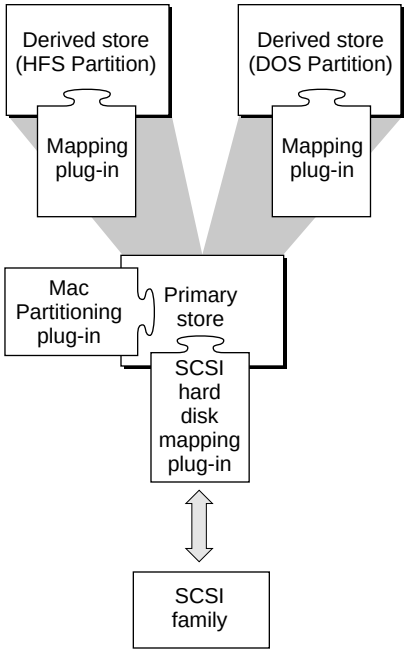
Partitioning Plug-ins

Partitioning plug-ins create, maintain, and extract information from a store's partition map.

Both primary and derived stores can have a partition map, and hence a partitioning plug-in, associated with them. Leaf stores consist of a single partition, do not contain a partition map, and do not have a partitioning plug-in associated with them.

As an example, assume you have a SCSI hard disk partitioned into Mac OS and DOS virtual devices. Figure 7-5 illustrates this example.

Figure 7-5 Simple partition example

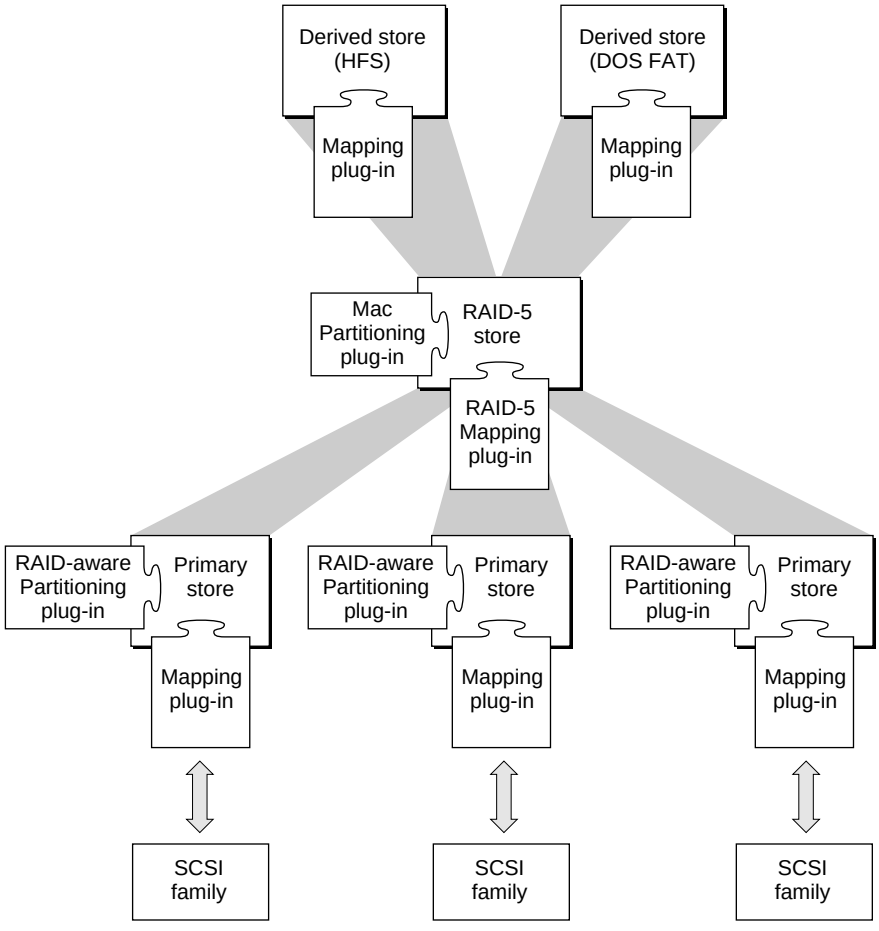


The bottom-most store in the figure is the primary store that maps to the SCSI hard disk. It has a SCSI hard disk mapping plug-in associated with it. The plug-in talks to the hard disk through the SCSI family. Because the store contains a Mac format partition map, the Apple-provided Mac partitioning plug-in is associated with it.

The two derived stores have the Apple-provided derived store mapping plug-in associated with them. Neither has a partitioning plug-in since both are leaf stores and neither has a partition map of its own.

Another interesting example is a software RAID-5 implementation. Figure 7-6 shows what the store hierarchy for such a device could look like:

Figure 7-6 RAID-5 partitioning



The RAID-5 driver is implemented as a mapping plug-in and is used at the derived store level. A RAID-aware partitioning plug-in stores the information needed by RAID-5 about its devices. The RAID-5 store is partitioned using the standard Mac partitioning format and contains an HFS and a DOS FAT partition, each of which results in a derived store.

Partitioning plug-ins are essential to Mac OS 8's auto-configuration. At system boot time, if a partitioning plug-in is associated with an existing store, the block

storage expert calls it to get information from the store's partition map and create a derived store for each partition. In this way, the expert automatically builds the entire store hierarchy.

Container Plug-ins

•••To be provided•••

Plug-in Discovery and Loading

Like all I/O plug-ins, block storage plug-ins must export a data structure of type `DriverDescription` named `TheDriverDescription`. The system needs this structure to find plug-ins, and match them to real and virtual devices. See "Driver and Family Matching" (page 2-3) for more information on how this process works and on the driver description structure.

The code for a mapping plug-in can be stored either in a folder named *Hardware Support* in the Mac OS Folder or in a Mac OS 8 mapping plug-in partition on a device. A plug-in located in a Mac OS 8 mapping plug-in partition on a device is always selected over a mapping plug-in in the folder. Drivers in a pre-Mac OS 8 driver partition are ignored.

Partitioning plug-ins and container plug-ins always reside in the Hardware Support folder.

The block storage expert works with Driver and Family Matching (DFM) software to recognize the presence of new devices and new CD-ROM discs, floppy disks, and other removable media, and to instantiate stores and containers. For more information on DFM, see "Driver and Family Matching" (page 2-3).

When a new block storage device is discovered, a low-level family expert creates a new node in the name registry with the following properties: the device type, the manufacturer name, and the version of the device. DFM matches plug-ins against these properties and adds to the device's node a list of pointers to the plug-ins that match. Then it notifies interested parties of the new device.

The block storage expert registers interest in new block storage device events. When it receives a new device notification, the expert creates a store, loads and initializes the correct mapping plug-in (and a partitioning plug-in if required), and initializes the store.

The block storage expert gives each matching mapping plug-in an opportunity to examine the device and read the media. Each plug-in reports a value that indicates how well it can support the device, based on the number of factors it recognizes. The higher the value, the better the match:

If the plug-in recognizes	The plug-in reports
the device type	1
the device manufacturer	2
the device version	3
the media	4

The expert selects the mapping plug-in reporting the highest value to manage the device.

If two or more plug-ins report the same value, then information in the `driverType` field in the plug-ins' driver description structures is used to select one.

The `driverType` field contains a `DriverType` structure. If two or more plug-ins have the same name in the `nameInfoStr` field of the `DriverType` structure, then the one with the latest version in the `version` field of the `DriverType` structure is selected. Otherwise, the values in the `nameInfoStr` field are sorted in ASCII order and the first one is selected.

Partitioning plug-in selection works differently from mapping plug-in selection. Instead of reporting defined levels of certainty, when a partitioning plug-in is given the opportunity to examine the store, it returns an integer. If the plug-in does not recognize the store's format, it should return 0. If it recognises the format, it returns the number of bytes it read from the store in determining that it recognises the format. The plug-in reading the greatest number of bytes is selected. If two plug-ins tie, the tie-breaking mechanism is the same as that described for mapping plug-ins.

Block Storage Family Activation Models

The block storage family activation models define the relationship between the family and its plug-ins—the mechanisms and conditions under which the plug-ins get called and what the plug-in's responses should be. The information in this section is of interest to plug-in developers. If you develop client software, you can read about it for general interest, but it should have no impact on how you write your code.

Activation Model For Mapping Plug-ins

The block storage family keeps track of a client's request, even if it is handled by several mapping plug-ins and utilizes several different devices and families. To help the family do this, mapping plug-ins must observe block storage family conventions related to requests and completion notifications.

When the block storage family gets a client I/O request, it calls the I/O function (page 7-66) of the mapping plug-in responsible for the target store and passes in a token that identifies the I/O request. That request is referred to as the *parent request*.

Note
Do not confuse multiple uses of the term parent. A parent request is not a request to a parent store. ◆

A mapping plug-in frequently needs to generate several I/O requests to another family or store in response to a single request that the plug-in gets from the block storage family. For example, assume a device can read or write a maximum of 512 bytes at a time.When the mapping plug-in for that device gets a request to read a larger number of bytes, it needs to break it up into a series of 512-byte requests.

Mapping plug-ins that manage primary stores call the `BSTrackOtherFamilyRequest` function (page 7-162) before calling another I/O family to handle an I/O request. A mapping plug-in passes these parameters to `BSTrackOtherFamilyRequest`:

- the token that identifies the parent request. The token enables the block storage family to call the correct plug-in's completion routine when the other I/O family signals that the request is complete. (Note that when a mapping plug-in makes several I/O requests in response to a single parent request, it passes the same token on each call to `BSTrackOtherFamilyRequest` for those I/O requests.)
- the plug-in's private data value. The block storage family passes this same value back to the plug-in when it calls the plug-in's completion routine for this I/O request.
- a pointer to a `KernelNotification` structure. The function fills in the structure to specify how the block storage family wants to be notified when the I/O request completes. The plug-in passes the `KernelNotification` structure to the I/O family that the plug-in calls to handle the I/O request.

After calling `BSTrackOtherFamilyRequest`, the plug-in can call another I/O family to service the request. Then, when it exits its I/O function, it should return the `kBSIOContinuing` result code.

Mapping plug-ins that manage derived stores call the `BSStoreRW` function (page 7-161) to make an I/O request to another plug-in. A mapping plug-in passes these parameters to `BSStoreRW`:

- the token that identifies the parent request
- the plug-in's private data value. The block storage family passes this same value back to the plug-in when it calls the plug-in's completion routine for this I/O request.

The block storage family then calls the I/O function of the mapping plug-in responsible for the target store and passes it the token.

A mapping plug-in's I/O completion routine is called once for each request it makes to another family or store. When all of the requests issued to satisfy a parent request are complete, the plug-in should return the `kBSIOCompleted` result code from its I/O completion routine for the parent request. That causes the block storage family to call the plug-in's I/O completion routine for the parent request.

A mapping plug-in does not care if the original client request was made synchronously or asynchronously. The block storage family uses an accept function to receive synchronous I/O requests from clients. As a result, a mapping plug-in executes in the client's context and can be executing in several contexts simultaneously. For more information on accept functions, see *Microkernel and Core System Services*.

To receive asynchronous I/O requests from clients, the block storage family creates a set of tasks when it initializes itself. A client's asynchronous request results in a microkernel message to the family. On receiving the message, the plug-in dispatches the request to one of its tasks. From that point, the request is handled inside the family in exactly the same way as a synchronous request.

The activation model for block storage mapping plug-ins results in the following requirments and guidelines.

- A mapping plug-in must be fully re-entrant and able to handle multiple pending I/O requests because it can be executing in several contexts simultaneously.
- Although a mapping plug-in can block while waiting on an event or acquiring a lock, it must avoid a deadlock situation. When it takes an action

that might cause it to be suspended, it must be certain that it will be reactivated. Allocating memory outside of the plug-in's initialization function is not recommended. Because a mapping plug-in is on the page fault path, it can cause deadlock if, to service a page fault, it causes a page fault.

- A mapping plug-in should be divided into two parts: a main code section that executes when it gets a request, and an I/O completion routine. This structure allows all of the main code sections involved in a request to start their I/O requests before any of the requests complete.
- A mapping plug-in should maintain one request queue per device. When it receives a request, the plug-in should dispatch it asynchronously to another family or store, or put it on the request queue. When the plug-in's completion routine is called for the request, it should dispatch the next request from the queue and return to block storage.

Note
Rather than simply adding a new request to the end of the queue, a plug-in can try to increase device throughput by inserting a new request at a spot in the queue that's optimal for the device. For example, by ordering requests based on a hard disk's tracks and sectors, a plug-in can reduce the mechanical head movement back and forth and get better performance from the device. ◆

The activation model for block storage mapping plug-ins described so far assumes the plug-in operates asynchronously—it starts an I/O request and returns to the block storage family, which in turn calls the plug-in's completion routine when the I/O is complete. However, synchronous operation may make more sense for a plug-in in some cases.

For example, if a mapping plug-in for a primary store does not use another family—such as if the store's data resides on a RAM disk—the plug-in should return the `kBSIOCompleted` result code from its I/O function. Because the plug-in can immediately move data between the RAM disk and a buffer in memory, there is nothing to wait on and no need for a completion routine.

When a mapping plug-in calls another plug-in (with the `BSStoreRW` function), it does not need to know if the called plug-in operates synchronously or asynchronously. It should assume an asynchronous plug-in. The block storage family manages things so that they work correctly, regardless of whether the plug-in in fact operates synchronously or asynchronously.

Note
Some developers might want to write synchronous mapping plug-ins that bypass the block storage mechanisms in an attempt to improve performance. Such a plug-in gets an I/O request and returns to block storage only when the request is complete. It must provide its own mechanisms for tracking multiple I/O requests when it splits one request into many, as well as the resulting notifications and completions. In essense, this design recreates within the plug-in much of the block storage family infrastructure. You should carefully consider the tradeoffs before making such a choice. Although it is often possible to get a small speed boost this way, you lose flexibility. If you bypass the block storage family tracking mechanisms, you can no longer freely mix and match your plug-in with others. For example, a synchronous RAID-aware mapping plug-in won't automatically work with another vendor's SCSI disk driver. In addition, the plug-in code becomes more complex and harder to test. ◆

Activation Model For Partitioning and Container Plug-ins

The model for partitioning and container plug-ins is simpler than that for mapping plug-ins, because partitioning and container plug-ins are not on the I/O path.

Partitioning and container plug-ins are always called synchronously in a task context and they are allowed to block.

Partitioning plug-ins call block storage client functions, such as `BSStoreConnRead` or `BSStoreConnWrite`, to make I/O requests to the family.

Block Storage Client Constants and Data Types

This section describes the data types and constants in the programming interface that the block storage family provides for its clients. A client, generally a file systems plug-in or a disk utility program, uses the block storage family to configure and access data on devices abstracted by the block storage programming interface.

Block Storage Byte Count Type

The block storage byte count data type provides a larger alternative to the system-wide defined type `ByteCount` that specifies a 32-bit value.

BSByteCount

The block storage family defines the `BSByteCount` data type to specify a 64-bit byte count integer. The Math64 library provides many support functions that you can use to manipulate values of type `BSByteCount`.

```
typedef UInt64 BSByteCount;
```

Block Storage ID Types

The data types in this section define persistent IDs. An ID is a unique value generated by Mac OS 8 that identifies a thing for as long as the thing exists, including across boots of Mac OS 8.

An ID can identify such things as a store, a container, or a connection to a store or a container.

BSStoreID

The `BSStoreCreate` function (page 7-114) returns a store ID. You can use the ID to get a reference to a store by calling the `BSStoreFindByID` function (page 7-112).

The block storage family defines the `BSStoreID` data type for a store ID.

```
typedef char BSStoreID[12];
```

BSStoreConnID

A **connection** is a logical path to a store and serves to control access to the store. Most clients obtain a connection to a store by calling the `BSStoreOpen` function (page 7-78). The function returns a connection ID which you can then pass to block storage functions to read and write to the store and to close it. This type of connection is called an **I/O connection** because it permits I/O to the store.

If your software creates and configures stores, you can obtain a connection ID for a store by calling the `BSStoreCreate` function (page 7-114). The function returns a connection ID which you can then pass to block storage functions to configure the store. This type of connection is called a **control connection** because it allows you to configure the store and to change its accessibility state, but it does not permit I/O to the store.

The block storage family defines the `BSStoreConnID` data type for a store connection ID.

```
typedef ObjectID BSStoreConnID;
```

BSContainerID

The block storage family defines the `BSContainerID` data type for a container ID.

```
typedef char BSContainerID[12];
```

BSContainerConnID

The block storage family defines the `BSContainerConnID` data type for a container connection ID.

```
typedef ObjectID BSContainerConnID;
```

Block Storage Reference Types

This section describes the run-time reference types defined by the block storage family. A reference is a unique identifier generated at run time. Unlike an ID, a reference cannot persist across boots of Mac OS 8.

BSStoreRef

You need a store reference to open a store and get a connection ID.

You can get a reference to a store by calling any of several store hierarchy navigation functions. See "Navigating a Store Hierarchy" (page 7-100) for more information.

The block storage family defines the `BSStoreRef` data type for a store run-time reference.

```
typedef RegEntryRef     BSStoreRef;
```

BSContainerRef

The block storage family defines the `BSContainerRef` data type for a container run-time reference.

```
typedef RegEntryRef     BSContainerRef;
```

BSMappingPlugInRef

The block storage family defines the `BSMappingPlugInRef` data type for a mapping plug-in run-time reference.

Plug-in code may be instantiated multiple times by the Code Fragment Manager, once for each store the plug-in is asked to support. The mapping plug-in reference identifies the plug-in code itself, not a particular instance of it.

```
typedef struct OpaqueBSMappingPlugInRef *BSMappingPlugInRef;
```

BSPartitioningPlugInRef

The block storage family defines the `BSPartitioningPlugInRef` data type for a partitioning plug-in run-time reference.

Plug-in code may be instantiated multiple times by the Code Fragment Manager, once for each store the plug-in is asked to support. The partitioning plug-in reference identifies the plug-in code itself, not a particular instance of it.

```
typedef struct OpaqueBSPartitioningPlugInRef *BSPartitioningPlugInRef;
```

BSContainerPlugInRef

The block storage family defines the `BSContainerPlugInRef` data type for a container plug-in run-time reference.

Plug-in code may be instantiated multiple times by the Code Fragment Manager, once for each store the plug-in is asked to support. The container plug-in reference identifies the plug-in code itself, not a particular instance of it.

```
typedef struct OpaqueBSContainerPlugInRef *BSContainerPlugInRef;
```

BSBlockListRef

A block list is an opaque data structure that specifies the address ranges in a store for a given I/O transfer. A single block list can accomodate a transfer of up to 4 gigabytes.

When you call the `BSBlockListCreate` function (page 7-81), it returns a reference to a new empty block list. To build and complete a block list, you pass the reference to the `BSBlockListAddRange` (page 7-82) and `BSBlockListFinalize` (page 7-84) functions. The block storage family defines the `BSBlockListRef` data type for a block list run-time reference.

```
typedef struct OpaqueBSBlockListRef *BSBlockListRef;
```

BSBlockListDescriptorRef

A block list descriptor is an opaque data structure that specifies a view of a block list.

When you call the `BSBlockListFinalize` (page 7-84) function, it returns a reference to a newly created block list descriptor. You use the descriptor when calling the following functions to read or write data: `BSStoreConnReadSG` (page 7-89), `BSStoreConnReadSGAsync` (page 7-91), `BSStoreConnWriteSG` (page 7-95), and `BSStoreConnWriteSGAsync` (page 7-96).

The block storage family defines the `BSBlockListDescriptorRef` data type for a block list descriptor run-time reference.

```
typedef struct OpaqueBSBlockListDescriptorRef *BSBlockListDescriptorRef;
```

Note
Mapping plug-ins use block list descriptor references extensively. A descriptor contains a bias and a set of address ranges. The bias is a value that is applied to the addresses in the descriptor to get the corresponding addresses in a given store.

If an I/O request must pass through more than one store, the mapping plug-ins can create additional descriptors, based on a specified existing descriptor. Descriptors eliminate the need to copy the block list itself and adjust the addresses it contains. Because you can create any number of descriptors for a single block list, it's also easy, in effect, to split a single block list into sublists by creating multiple descriptors.

For information about the functions mapping plug-in use with block list descriptors, see "Working With a Block List Descriptor" (page 7-147). ◆

Navigation Types

The block storage family maintains a hierarchy of stores and of containers. For information on the store hierarchy, see "Stores" (page 7-13).

The block storage family provides functions, described in "Navigating a Store Hierarchy" (page 7-100) and "Navigating a Container Hierarchy" (page 7-131), that allow you to navigate a store and container hierarchies and retrieve properties.

The navigation functions use the data types described in this section.

BSStoreGetSelector

When you call the `BSStoreGetDeviceData` function (page 7-100), you indicate the set of stores about which you want information.

The block storage family defines the `BSStoreGetSelector` data type and its enumerated values for store set selectors.

```
typedef UInt32  BSStoreGetSelector;

enum {
    kBSStoreGetLeafStores = 1,
    kBSStoreGetAllStores = 2,
    kBSStoreGetPrimaryStores =3
};
```

Enumerator descriptions

`kBSStoreGetLeafStores`
: Returns information on all leaf stores.

`kBSStoreGetAllStores`
: Returns information on all stores.

`kBSStoreGetPrimaryStores`
: Returns information on all primary stores.

BSStoreIOIteratorData

When you call the `BSStoreGetDeviceData` function (page 7-100), the function returns information about one or more stores.

The block storage family defines the `BSStoreIOIteratorData` structure to contain the store information that `BSStoreGetDeviceData` can return.

```
struct BSStoreIOIteratorData {
    IOCommonInfo        IOCI;
    BSStoreInfo         info;
};
```

Field descriptions

`IOCI` A structure of type `IOCommonInfo`. It contains a reference number that, within the block storage family, uniquely identifies the store described by the `info` field. It also contains the version of this `BSStoreIOIteratorData` structure. See "About the I/O Architecture" (page 1-3) for information on the `IOCommonInfo` type.

`info` A store information structure (page 7-44).

BSStoreIteratorID

To browse a store hierarchy, you need an iterator—an opaque value that points to a node in the hierarchy. It defines the point from which a navigation of the hierarchy begins.

You create an iterator with the `BSStoreIteratorCreate` function (page 7-101) and then use it with the functions described in "Navigating a Store Hierarchy" (page 7-100).

The block storage family defines the `BSStoreIteratorID` data type for a store hierarchy iterator.

```
typedef struct opaque BSStoreIteratorID;
```

BSStorePropertyInstance

An instance value for a property is assigned by the block storage family when the instance is created. An instance value distinguishes a property instance from other instances of properties with the same name. It is unique and it persists for the life of the property or until the system is shut down. (New instance values are assigned to restored persistent properties at system startup.

Store property instances start at 0 and increment by 1. You can use the instance value to get the size and actual value of the property by calling the `BSStoreGetPropertySize` (page 7-110) and `BSStoreGetProperty` (page 7-111) functions .

The block storage family defines the `BSStorePropertyInstance` data type for a store property instance value.

```
typedef  RegPropertyInstance BSStorePropertyInstance;
```

BSContainerIteratorID

To browse a container hierarchy, you need an iterator—an opaque value that points to a node in the hierarchy. It defines the point from which a navigation of the hierarchy begins.

You create an iterator with the `BSContainerIteratorCreate` function (page 7-131) and then use it with the functions described in "Navigating a Container Hierarchy" (page 7-131).

The block storage family defines the `BSContainerIteratorID` data type for a container hierarchy iterator.

```
typedef  ObjectID BSContainerIteratorID;
```

BSContainerPropertyInstance

An instance value for a property is assigned by the block storage family when the instance is created. An instance value distinguishes a property instance

from other instances of properties with the same name. It is unique and it persists for the life of the property or until the system is shut down. (New instance values are assigned to restored persistent properties at system startup.

Container property instances start at 0 and increment by 1. You can use the instance value to get the size and actual value of the property by calling the `BSContainerGetPropertySize` (page 7-137) and `BSContainerGetProperty` (page 7-138) functions .

The block storage family defines the `BSContainerPropertyInstance` data type for a container property instance value.

```
typedef  RegPropertyInstance BSContainerPropertyInstance;
```

Store Property Names

You provide a store property name to identify the property of interest when you call the `BSStoreGetPropertySize` (page 7-110) and `BSStoreGetProperty` (page 7-111) functions. The following store property names are defined:

```
#define kBlockStorageStoreID                "Id"
#define kBlockStorageStoreSize              "BSStoreSize"
#define kBlockStorageStoreReadBlockSize     "BSStoreReadBlockSize"
#define kBlockStorageStoreWriteBlockSize    "BSStoreWriteBlockSize"
#define kBlockStorageStoreContainer         "BSStoreContainer"
#define kBlockStorageStoreParent            "BSStoreParent"
#define kBlockStorageStoreType              "BSStoreType"
#define kBlockStorageStoreDevice            "BSStoreDevice"
#define kBlockStorageEjectable              "BSStoreEjectable"
#define kBlockStorageBootDevice             "BSStoreBootDevice"
#define kBlockStorageWritable               "BSStoreWritable"
#define kBlockStorageMappingPlugIn          "BSStoreMappingPlugIn"
#define kBlockStoragePartitioningPlugIn     "BSStorePartitioningPlugIn"
```

Container Property Names

You provide a container property name to identify the property of interest when you call the `BSContainerGetPropertySize` (page 7-137) and `BSContainerGetProperty` (page 7-138) functions. The following container property names are defined:

```
#define kBlockStorageContainerParent        "BSStoreParent"
#define kBlockStorageContainerType          "BSStoreType"
#define kBlockStorageEjectable              "BSStoreEjectable"
#define kBlockStorageContainerPlugIn        "BSStoreContainerPlugIn"
```

Store Format Types

The data types described in this section provide information about the format of a store.

BSStoreFormatType

The block storage family defines the `BSStoreFormatType` data type and its enumerated values to designate the format type of a store's media. The format types describe broad categories.

The `BSStoreFormatType` type is used in the structure type `BSStoreFormatInfo` (page 7-38).

```
typedef OSType BSStoreFormatType;

enum {
    kBSFormatFloppyGCR  = 'gcr ',
    kBSFormatFloppyMFM  = 'mfm ',
    kBSFormatSCSI       = 'scsi',
    kBSFormatATA        = 'ata ',
    kBSNotFormatable    = 'none',
};
```

Enumerator descriptions

`kBSFormatFloppyGCR` The Group Code Recording format used on 800 KB 3.5 inch floppy disks.

`kBSFormatFloppyMFM` The Modified Frequency Modulation format used on 1.44 MB 3.5 inch floppy disks.

`kBSFormatSCSI` The format used on SCSI hard disks.

`kBSFormatATA` The format used on ATA hard disks.

kBSNotFormatable	The media cannot be formatted (for example, a CD-ROM disc).

BSFormatIndex

Clients who configure stores can call the BSStoreConnFormat function (page 7-125) to format a store. The block storage family defines the BSFormatIndex data type to identify the format to be used.

Values of type BSFormatIndex are specific to a given mapping plug-in; they do not intrinsically identify a particular format. Rather, they associate a specific format type (identified by BSStoreFormatType (page 7-37)) with an index value specific to the plug-in. Index values start at 0.

The BSFormatIndex type also forms part of the structure type BSStoreFormatInfo (page 7-38).

```
typedef ItemCount BSFormatIndex;
```

BSStoreFormatInfo

The block storage family defines the BSStoreFormatInfo data type to provide information about a format that can be supported by a mapping plug-in. A BSStoreFormatInfo structure forms part of the structure type BSStoreInfo (page 7-44).

```
struct BSStoreFormatInfo {
    BSStoreFormatType   formatType;
    BSByteCount         formatSize;
    BSFormatIndex       formatNum;
};
```

Field descriptions

formatType	A value of type BSStoreFormatType (page 7-37) that specifies a format type.
formatSize	The number of bytes that are available for I/O with this format type. This value distinguishes subtypes within a

given format type. For example, format type `kBSFormatFloppyMFM` indicates a 1.44 MB floppy disk. Such a disk can be one of several subtypes, differing in the number of bytes available for I/O on the disk, such as 1.44 MB, 1 MB, and so forth. If a given format type has no subtypes or if the `formatType` field is set to `kBSNotFormatable`, this field contains 0.

`formatNum` An index value, specific to the mapping plug-in associated with a given store, that identifies the format.

Maximum Formats Constant

The `possibleFormats` field of the `BSStoreInfo` structure (page 7-44) contains an array of `BSStoreFormatInfo` structures (page 7-38). The maximum size of the array is defined by the `kBSMaxFormats` constant.

```
enum { kBSMaxFormats = 8 };
```

Store Component Types

A component is a piece of a store, either a device, another store, or a partition of a store. Most stores have only one component. Some stores may have multiple components.

The data types in this section define components.

BSComponentType

The block storage family defines the `BSComponentType` data type and enumerated values for types of components.

The `BSComponentType` type is part of the `BSStoreComponent` structure type (page 7-40).

```
typedef UInt32      BSComponentType;
```

```
enum {      /* values of BSComponentType */
    kBSExternalDeviceComponent  = 1,    /* physical device */
    kBSStoreComponent           = 2     /* virtual device */
};
```

Enumerator descriptions

`kBSExternalDeviceComponent`
: The component is a physical device or a portion of a physical device.

`kBSStoreComponent`
: The component is another store.

BSStoreComponent

When you call the `BSStoreConnGetComponents` (page 7-124) function, the block storage family returns information about the components of a store in a store component structure, defined by the `BSStoreComponent` data type.

```
struct BSStoreComponent {
    BSComponentType         componentType;  /* physical or logical */
    BSByteCount             startingOffset;
    RegEntryRef             sourceNode;     /* for physical components */
    BSStoreRef              srcStore;       /* for logical components */
    BSPartitionDescriptor   partitionInfo;  /* for logical components */
};

typedef BSStoreComponent *BSStoreComponentPtr;
```

Field descriptions

`componentType`
: A value that indicates whether the component is a physical device or another store (virtual device). See "BSComponentType" (page 7-39) for a listing of component types.

`startingOffset`
: The number of bytes into the store at which this component begins. A store always begins at byte 0.

`sourceNode`
: If the `componentType` field contains `kBSExternalDeviceComponent`, this field contains a reference

	to the name registry entry that identifies a physical device. Otherwise, this field contains 0.
`srcStore`	If the `componentType` field contains `kBSStoreComponent`, this field contains a reference to the store that is the logical component. Otherwise, this field contains 0.
`partitionInfo`	If the `componentType` field contains `kBSStoreComponent`, this field contains a `BSPartitionDescriptor` structure (page 7-48) that provides information about the partition map of this component. Otherwise, this field contains 0.

Note
The `BSStoreMPIComponent` data type (page 7-57) used by mapping plug-ins has the same definition as the `BSStoreComponent` data type. ◆

Accessibility State Type

•••To be provided•••

BSAccessibilityState

An accessibility state indicates how readily a block storage container or store can be accessed. You can change the accessibility state of containers and stores with the `BSContainerConnGoToAccessibilityState` (page 7-130) and `BSStoreConnGoToAccessibilityState` (page 7-99) functions. To specify an accessibility state, the block storage family defines the `BSAccessibilityState` data type and its enumerated values.

```
typedef UInt32 BSAccessibilityState;

enum {
    kBSOnline = 0x0FFFFFFF,
    kBSPowerSave = 0x0A000000,
    kBSOutOfDrive = 0x01000000,
    kBSOffline = 0,
};
```

Constant descriptions

`kBSOnline`	The container or store is immediately available.
`kBSPowerSave`	The container or store is not immediately available because the device it maps to is in a power-save mode. A container or store in this state is more accessible than one in the `kBSOutOfDrive` state.
`kBSOutOfDrive`	The container or store is not immediately available because the media is not in a drive, although it remains under programmatic control. A container or store in this state (for example, a CD disc in a CD-ROM autochanger) is less accessible than one in the `kBSPowerSave` state.
`kBSOffline`	The container or store is not available because the device it maps to is no longer under programmatic control. This is the least accessible state. When you put a container or store into this state, the block storage family deletes it from the store hierarchy and causes the mapping plug-in to eject the corresponding media from the device.

Open Options Types

•••To be provided•••

BSStoreOpenOptions

When you open a connection to a store with the `BSStoreOpen` function (page 7-78), you supply options that define the type of connection you want. If the function returns with no error, you have a connection with the characteristics you requested. Connection types are described in "BSStoreConnID" (page 7-29).

The block storage family defines the `BSStoreOpenOptions` data type and its enumerated values for open options.

```
typedef UInt32 BSStoreOpenOptions;
```

```
enum {
    kBSStoreRead            = 0x00000001L,  /* allow reading */
    kBSStoreWrite           = 0x00000002L,  /* allow writing */
    kBSStoreExclusiveIO     = 0x00000004L,  /* don't allow another I/O connection */
    kBSStoreExclusiveCntrl  = 0x00000008L,  /* don't allow more control connections */
    kBSStoreResizeOK        = 0x00000010L,  /* reserved */
    kBSStoreControl         = 0x00000020L,  /* allow control of the store */
};
```

Constant descriptions

`kBSStoreRead` An I/O connection that allows reads from the store.

`kBSStoreWrite` An I/O connection that allows writes to the store.

`kBSStoreExclusiveIO`
An exclusive I/O connection. If granted, other I/O connection requests to the store are denied until this connection is closed.

`kBSStoreExclusiveCntrl`
An exclusive control connection. If granted, other control connection requests to the store are denied until this connection is closed.

`kBSStoreResizeOK` Reserved. Always 0.

`kBSStoreControl` A control connection that allows the store to be configured.

The `kBSStoreRead`, `kBSStoreWrite`, and `kBSStoreControl` options each specify a limited and mutually exclusive set of operations that can be performed on a store once the connection is granted. For example, if you specify only the `kBSStoreRead` option, you can read the store, but you cannot write to it or configure it. However, You can OR the enumerated option values together in any combination to specify connection types such as read/write, control/exlusive, and so on.

BSContainerOpenOptions

When you open a connection to a container with the `BSContainerOpen` function (page 7-128), you supply options that define the access you wish to allow to the container.

The block storage family defines the `BSContainerOpenOptions` data type and an enumerated value.

```
typedef UInt32 BSContainerOpenOptions;

enum {
    kBSContainerExclusiveCntrl  = 0x00000001,
};
```

Constant description

`kBSContainerExclusiveCntrl`
Don't allow another connection to be opened to the container.

Store Information Structure

•••To be provided•••

BSStoreInfo

When you call the `BSStoreConnGetInfo` function (page 7-123), the block storage family returns information about a store in a store information structure, defined by the `BSStoreInfo` data type.

A store information structure is also part of the `BSStoreIOIteratorData` type (page 7-34).

```
struct BSStoreInfo {
    BSStoreID       storeID;        /* store ID */
    BSByteCount     storeSize;      /* number of bytes in the store */
    BSByteCount     readBlockSize;  /* minimum read size & granularity */
    BSByteCount     writeBlockSize; /* minimum write size & granularity */
    BSContainerRef  container;      /* the container containing this store */
    ItemCount       numChildren;    /* number of stores derived from this store */
    ItemCount       numParents;     /* number of parent stores */
    ItemCount       numPartitions;  /* number of partitions in use */
    ItemCount       maxPartitions;  /* maximum number of partitions possible */
    Boolean         isPartitioned;  /* is partitioning plug-in associated with
```

```
                                        store? */
    Boolean         isEjectable;
    Boolean         isBootDevice;
    Boolean         isWriteable;
    Boolean         hasAutoEjectHardware;
    Boolean         isFormattable;
    Boolean         isPartitionable;
    Boolean         isFilesystem;
    BSMappingPlugInRef          mappingPlugIn;
    BSPartitioningPlugInRef     partitioningPlugIn;
    BSStoreFormatInfo           curFormat;
    BSStoreFormatInfo           possibleFormats [kBSMaxFormats];
    Str255          name;           /* name of store */
    Str32           typeName;       /* name of partition type */
};

            typedef BSStoreInfo *BSStoreInfoPtr;
```

Field descriptions

`storeID`	The ID of the store. The ID is a unique value generated by Mac OS 8 that identifies the store for as long as it exists, even across system restarts.
`storeSize`	The size of the store, in bytes.
`readBlockSize`	The minimum size, in bytes, of a read request to the store. All read requests to the store must use a multiple of the read block size.
`writeBlockSize`	The minimum size, in bytes, of a write request to the store. All write requests to the store must use a multiple of the write block size.
`container`	A reference to the container that contains the store.
`numChildren`	The number of stores derived from this store.
`numParents`	The number of parent stores to which this store maps. For primary stores, this field contains 0.
`numPartitions`	The number of partitions defined for this store.
`maxPartitions`	The maximum number of partitions it is possible to define for this store.
`isPartitioned`	A Boolean value. The value `true` indicates that a partitioning plug-in is associated with this store.

`isEjectable`	A Boolean value. The value `true` indicates that this store maps to media that can be ejected.
`isBootDevice`	A Boolean value. The value `true` indicates that this store maps to the boot device.
`isWriteable`	A Boolean value. The value `true` indicates that this store allows write operations.
`hasAutoEjectHardware`	A Boolean value. The value `true` indicates that this store maps to a device with auto-eject hardware, such as some types of floppy disk drives.
`isFormattable`	A Boolean value. The value `true` indicates that this store allows formatting operations.
`isPartitionable`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the expert should attempt to match a partitioning plug-in with this store.
`isFilesystem`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the File Manager should be notified to attempt to match a file system plug-in with this store
`mappingPlugIn`	A reference to the mapping plug-in associated with this store.
`partitioningPlugIn`	A reference to the partitioning plug-in associated with this store.
`curFormat`	A `BSStoreFormatInfo` structure (page 7-38) containing information about the format currently in use for this store.
`possibleFormats`	An array of `BSStoreFormatInfo` structures (page 7-38) containing information about the formats that the mapping plug-in for this store can support.
`name`	A pointer to a null-terminated string of ASCII characters that specify the user-readable name of this store. For derived stores, this is the name of the partition it maps to in the parent store. The block storage family provides the names for primary stores.
`typeName`	A pointer to a null-terminated string of ASCII characters that specify the partition type of this store—for example, “Apple_HFS”.

Note
A mapping plug-in's add component function (page 7-187) updates the information in a store information structure.
◆

Container Information Structure

•••To be provided•••

BSContainerInfo

The block storage family returns information about a container in a container information structure when you call the `BSContainerConnGetInfo` function (page 7-142). A container information structure is defined by the `BSContainerInfo` data type.

```
struct BSContainerInfo {
    RegEntryRef     device;
    ItemCount       numChildren;
    Boolean         ejectable;
};

typedef BSContainerInfo *BSContainerInfoPtr;
```

Field descriptions

`device`	The name registry reference for the device that this container represents.
`numChildren`	The number of name registry child nodes that belong to this container.
`ejectable`	A Boolean value. The value `true` indicates that the device is ejectable.

Partition Descriptor Structure

A partition is a portion of a store that is allocated to a particular operating system, file system, or device driver. Each store contains one or more partitions.

A partition descriptor structure contains basic information about a partition. It is used by formatting software when configuring stores.

BSPartitionDescriptor

When you call the `BSStoreConnGetPartitionInfo` function (page 7-119), the function returns information about a partition in a partition descriptor structure. When you call the `BSStoreConnSetPartitionInfo` function (page 7-118), you supply basic partition information.

The block storage family defines the `BSPartitionDescriptor` structure to contain basic partition information that is typically available, regardless of the partition format.

```
struct BSPartitionDescriptor {
    ItemCount       entryNum;           /* mandatory */
    BSByteCount     start;              /* mandatory */
    BSByteCount     len;                /* mandatory */
    BSStoreID       destStoreID;        /* optional */
    Boolean         isPartitionable;
    Boolean         isFilesystem;
    Boolean         reserved1;
    Boolean         reserved2;
    Str255          name;               /* optional */
    Str32           typeName;           /* optional */
};

typedef BSPartitionDescriptor   *BSPartitionDescriptorPtr;
```

Field descriptions

`entryNum` The ordinal number of the partition. The first partition is numbered 0, the second partition is numbered 1, and so on.

`start` The byte address within the store at which the partition starts. Stores addresses start at 0.

`len`	The size of the partition, in bytes.
`destStoreID`	The ID of the store that corresponds to this partition. This field contains 0 if the corresponding store has not yet been created and configured.
`isPartitionable`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the expert should attempt to match a partitioning plug-in with the derived store that is based on this partition.
`isFilesystem`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the File Manager should be notified to attempt to match a file system plug-in with the store that is based on this partition.
`reserved1`	Reserved.
`reserved2`	Reserved.
`name`	A pointer to a null-terminated string of ASCII characters that specify the user-readable name of the partition. If the name is not available, this field contains 0.
`typeName`	A pointer to a null-terminated string of ASCII characters that specify the user-readable type of the partition, for example, Apple HFS. If the type is not available, this field contains 0

Block Storage Plug-in Constants and Data Types

This section describes the data types and constants in the programming interface that the block storage family provides for its plug-ins. The block storage family defines 3 types of plug-ins:

- mapping plug-ins
- partitioning plug-ins
- container plug-ins

Some elements of the plug-in programming interface are common to all plug-in types and some are specific to a given type. The description of a data type indicates the type of plug-in it applies to.

I/O Constants

The block storage family defines the following constants for specifying a read or write operation. They are used in the `options` parameter of a mapping plug-in's I/O function (page 7-183) and of the `BSStoreRW` function (page 7-161).

```
enum {
    kBSRead = 0,
    kBSWrite = 1
};
```

Basic Block Storage Types For Use By Plug-ins

The data types in this section all point to opaque data types.

BSStorePtr

The block storage family uses an opaque data type to implement a store and defines the `BSStorePtr` data type to point to it.

```
typedef void *BSStorePtr;
```

BSContainerPtr

The block storage family uses an opaque data type to implement a container and defines the `BSContainerPtr` data type to point to it.

```
typedef void *BSContainerPtr;
```

BSIORequestBlockPtr

The block storage family defines the `BSIORequestBlockPtr` data type to point to a token that identifies an I/O request.

```
typedef void *BSIORequestBlockPtr;
```

Block List Descriptor Types

The data types in this section provide information about block list descriptors. A client that needs to read or write to a store creates a block list and gets a block list descriptor reference to pass with its I/O request. A mapping plug-in uses a block list descriptor reference when implementing an I/O request. See "Building a Block List" (page 7-80) and "Working With a Block List Descriptor" (page 7-147) for additional information.

BSBlockListDescriptorInfo

When you call the `BSBlockListDescriptorGetInfo` (page 7-147) function, the block storage family returns information about a block list descriptor. A block list descriptor information structure is defined by the `BSBlockListDescriptorInfo` data type.

```
struct BSBlockListDescriptorInfo {
    BSByteCount                bias;
    BSByteCount                start;
    UInt32                     length;
    BSBlockListDescriptorRef   parentDescriptor;
    BSBlockListRef             parentList;
};

typedef BSBlockListDescriptorInfo *BSBlockListDescriptorInfoPtr;
```

Field descriptions

`bias` The number of bytes the addresses in this descriptor are shifted from the addresses in the parent descriptor. If there is no parent descriptor, this field contains 0. Once a descriptor is created, the bias never changes. A negative bias is not valid.

`start` The number of bytes into the parent descriptor at which this descriptor begins. If there is no parent descriptor, this field contains 0.

`length`	The number of bytes described by this descriptor.
`parentDescriptor`	A reference to the parent block list descriptor (page 7-32). If there is no parent descriptor, this field contains `null`.
`parentList`	A reference to the block list (page 7-31) that this descriptor describes.

BSBlockListWhence

When you call the `BSBlockListDescriptorSeek` (page 7-155) function to change the offset within a block list descriptor, you specify the method to be used when computing the new offset. The block storage family defines the `BSBlockListWhence` data type and its enumerated values for seek methods.

```
typedef UInt32  BSBlockListWhence;

enum {
    kBSBlockListSeekByteAbsolute = 1,
    kBSBlockListSeekByteRelative = 2,
    kBSBlockListSeekExtentAbsolute = 3,
    kBSBlockListSeekExtentRelative = 4,
    kBSBlockListSeekBlockAbsolute = 5,
    kBSBlockListSeekBlockRelative = 6
};
```

Enumerator descriptions

`kBSBlockListSeekByteAbsolute`
Set the offset to the address specified in the `BSBlockListDescriptorSeek` function's `offset` parameter. A negative address is not valid.

`kBSBlockListSeekByteRelative`
Compute the offset by adding the value in the `BSBlockListDescriptorSeek` function's `offset` parameter to the current offset.

`kBSBlockListSeekExtentAbsolute`
Set the offset to the beginning of the *nth* extent in the descriptor. Extent numbering starts at 0. The value *n*

specified in the `offset` parameter is interpreted as an ordinal. A negative *n* is not valid.

`kBSBlockListSeekExtentAbsolute`
Compute the offset by adding the value in the `offset` parameter to the ordinal value of the extext at the current offset.

`kBSBlockListSeekBlockAbsolute`
•••To be provided•••

`kBSBlockListSeekBlockRelative`
•••To be provided•••

Confidence Level Types

A confidence level indicates the degreee of confidence a plug-in has in its ability to support a given store.

BSMPIConfidenceLevel

When its examine function (page 7-179) is called, a mapping plug-in returns a confidence level that indicates how well it can support a given store. See "Plug-in Discovery and Loading" (page 7-22) for more information.

The block storage family defines the `BSMPIConfidenceLevel` data type and its enumerated values for mapping plug-in confidence levels.

```
typedef UInt32 BSMPIConfidenceLevel;

enum {
    kBSMPIDeviceNotSupported    = 0,
    kBSMPIDeviceTypeRecognized  = 1,
    kBSMPIDeviceMfrRecognized   = 2,
    kBSMPIDeviceModelRecognized = 3,
    kBSMPIDeviceMediaRecognized = 4,
};
```

Enumerator descriptions

`kBSMPIDeviceNotSupported`
: The mapping plug-in does not recognize the store, and thererfore cannot support it.

`kBSMPIDeviceTypeRecognized`
: The mapping plug-in recognizes the type of device.

`kBSMPIDeviceMfrRecognized`
: The mapping plug-in recognizes the type of device and its manufacturer.

`kBSMPIDeviceModelRecognized`
: The mapping plug-in recognizes the type of device, its manufacturer, and the device model.

`kBSMPIDeviceMediaRecognized`
: The mapping plug-in recognizes the type of device, its manufacturer and device model, and the media used by the device. This is the highest level of confidence that a plug-in can report about its ability to support a given device.

BSCPIConfidenceLevel

A container plug-in returns a confidence level when its examine function (page 7-198) is called. The block storage family defines the `BSCPIConfidenceLevel` data type and its enumerated values for container plug-in confidence levels.

```
typedef UInt32 BSCPIConfidenceLevel;

enum {
    kBSCPIDeviceNotSupported    = 0,
    kBSCPIDeviceTypeRecognized  = 1,
    kBSCPIDeviceMfrRecognized   = 2,
    kBSCPIDeviceModelRecognized = 3,
};
```

Enumerator descriptions

`kBSCPIDeviceNotSupported`
The container plug-in does not support the device.

`kBSCPIDeviceTypeRecognized`
The container plug-in recognizes the type of device and can support it.

`kBSCPIDeviceMfrRecognized`
The container plug-in recognizes the type of device and its manufacturer.

`kBSCPIDeviceModelRecognized`
The container plug-in recognizes the type of device, its manufacturer, and the device model.

Status and Error Types

•••To be provided•••

BSIOStatus

The `BSStoreRW` function (page 7-161) and a mapping plug-in's I/O function (page 7-183) return an I/O status code. The block storage family defines the `BSIOStatus` data type and its enumerated values for I/O status codes.

```
typedef UInt32 BSIOStatus;

enum {
    kBSIOCompleted      = 1,
    kBSIOContinuing     = 2,
    kBSIOFailed         = 3,
    kBSIONotStarted     = 4,
};
```

Enumerator descriptions

`kBSIOCompleted` The I/O request is complete. No further action is necessary.

`kBSIOContinuing`	The I/O request has started successfully. The block storage family will call the associated I/O completion routine when the request has completed.
`kBSIOFailed`	The I/O request failed.
`kBSIONotStarted`	The I/O request could not be started.

BSIOErrors

The block storage family defines the `BSIOErrors` data type for reporting an error on executing an I/O request.

The `BSIOErrors` type forms part of the structure type `BSErrorList` (page 7-56).

```
typedef OSStatus BSIOErrors;
```

BSErrorList

The block storage family defines the `BSErrorList` structure for reporting an error on executing an I/O request.

When a mapping plug-in needs to generate more than one I/O request to service a single I/O request that it gets from block storage, the error list allows a mapping plug-in to keep track of the current state of each I/O request it made. When a request completes successfully, a mapping plug-in's I/O completion routine is called with a `nil` error list pointer. However, if an error occurs, the completion routine gets one or more `BSErrorList` structures in a linked list, specifying the exact state of each transfer request.

```
struct BSErrorList {
    BSByteCount     startingBlock;
    BSByteCount     length;
    UInt32          status;
    BSIOErrors      error;
    ItemCount       xferID;
    BSErrorList     *next;
};
```

```
typedef BSErrorList *BSErrorListPtr;
```

Field descriptions

`startingBlock`	The initial transfer byte of this fragment of the request.
`length`	The number of bytes to be transfered, including the starting byte.
`status`	A value indicating the final state of the transfer.
`error`	The exact error, if any.
`xferID`	An index representing which request of a multiple-request I/O this error applies to.
`next`	A pointer to the next error list structure.

Store Component Type

•••To be provided•••

BSStoreMPIComponent

When a mapping plug-in calls the `BSStoreGetComponent` function (page 7-171) or when its add component function (page 7-187) is called, a `BSStoreMPIComponent` structure is passed as a parameter.

The `BSStoreMPIComponent` structure has the same definition as the `BSStoreComponent` structure (page 7-40).

Store Information Structures

The structures in this section each describe certain characteristics of a store. The type of information provided varies with the type of plug-in providing it.

BSStoreMPIInfo

When a mapping plug-in calls the `BSStoreGetMPIInfo` function (page 7-159) or when its information function (page 7-190) is called, a `BSStoreMPIInfo` structure is passed as a parameter.

The `BSStoreMPIInfo` structure has many of the same fields as the `BSStoreInfo` structure (page 7-44).

```
struct BSStoreMPIInfo {
    BSAccessibilityState    curState;
    BSByteCount             storeSize;
    BSByteCount             readBlockSize;
    BSByteCount             writeBlockSize;
    Boolean                 isEjectable;
    Boolean                 isWriteable;
    Boolean                 hasAutoEjectHardware;
    Boolean                 isFormattable;
    Boolean                 isPartitionable;
    Boolean                 isFilesystem;
    Boolean                 reserved1;
    Boolean                 reserved2;
    BSStoreFormatInfo       curFormat;
    BSStoreFormatInfo       possibleFormats [kBSMaxFormats];
    char                    name [kRegMaxEntryNameLength + 1];
};

typedef BSStoreMPIInfo *BSStoreMPIInfoPtr;
```

Field descriptions

`curState` The current accessibility state of the store.

`storeSize` The size of the store, in bytes.

`readBlockSize` The minimum size, in bytes, of a read request to the store. All read requests to the store must use a multiple of the read block size.

`writeBlockSize` The minimum size, in bytes, of a write request to the store. All write requests to the store must use a multiple of the write block size.

`isEjectable` A Boolean value. The value `true` indicates that this store maps to media that can be ejected.

`isBootDevice`	A Boolean value. The value `true` indicates that this store maps to the boot device.
`isWriteable`	A Boolean value. The value `true` indicates that this store allows write operations.
`hasAutoEjectHardware`	A Boolean value. The value `true` indicates that this store maps to a device with auto-eject hardwaree, such as some types of floppy disk drives.
`isFormattable`	A Boolean value. The value `true` indicates that this store allows formatting operations.
`isPartitionable`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the expert should attempt to match a partitioning plug-in with this store.
`isFilesystem`	A Boolean value that provides a hint to the block storage expert. The value `true` indicates that the File Manager should be notified to attempt to match a file system plug-in with this store
`reserved1`	Reserved.
`reserved2`	Reserved.
`curFormat`	A `BSStoreFormatInfo` structure (page 7-38) containing information about the format currently in use for this store.
`possibleFormats`	An array of `BSStoreFormatInfo` structures (page 7-38) containing information about the formats that the mapping plug-in for this store supports.
`name`	An arrary of ASCII characters that specify the user-readable name of this store.

BSStorePPIInfo

When a partitioning plug-in calls the `BSStoreGetPPIInfo` function (page 7-160) or when its information function (page 7-195) is called, a `BSStorePPIInfo` structure is passed as a parameter.

```
struct BSStorePPIInfo {
    ItemCount   numPartitions;
    ItemCount   maxPartitions;
};

typedef BSStorePPIInfo *BSStorePPIInfoPtr;
```

Field descriptions

`numPartitions` The number of partitions currently configured for a store.

`maxPartitions` The maximum number of partitions the store can support.

Container Information Type

•••To be provided•••

BSContainerPIInfo

A container plug-in's information function (page 7-202) returns information about a container in a `BSContainerPIInfo` structure.

```
struct BSContainerPIInfo {
    ItemCount       numChildren;
    Boolean         isEjectable;
};

typedef BSContainerPIInfo *BSContainerPIInfoPtr;
```

Field descriptions

`numChildren` **•••To be provided•••**

`isEjectable` **•••To be provided•••**

Plug-in Interface Version Constant

The `kBSPlugInInterfaceVersion` constant indicates the version of the block storage plug-in interface used by a plug-in. A plug-in sets the `version` field of its `BlockStoragePlugInInfo` structure (page 7-61) at compile time.

```
enum {kBSPlugInInterfaceVersion = 0x02011996};
```

Plug-in Interface Structures

The structures in this section are the means by which a block storage plug-in provides the block storage family with the entrypoints to its functions.

BlockStoragePlugInInfo

A `BlockStoragePlugInInfo` structure contains the version of the block storage plug-in interface used by a plug-in.

It is a substructure in the `BSStoreMappingOps` (page 7-62), `BSStorePartitioningOps` (page 7-63), and `BSContainerPolicyOps` (page 7-64) structures.

```
struct BlockStoragePlugInInfo {
    UInt32  version;
    UInt32  reserved1;
    UInt32  reserved2;
    UInt32  reserved3;
};
```

Field descriptions

`version` The version of the block storage plug-in interface used by this plug-in. Set this field to the constant `kBSPlugInInterfaceVersion`, described in "Plug-in Interface Version Constant" (page 7-61).

`reserved1` Reserved.

`reserved2` Reserved.

`reserved3` Reserved.

BSStoreMappingOps

A `BSStoreMappingOps` structure contains the entrypoints of the functions a mapping plug-in provides to the block storage family.

A mapping plug-in must export a structure of type `BSStoreMappingOps` named `BlockStorageMappingPIOps` to the block storage family.

```
struct BSStoreMappingOps {
    BlockStoragePlugInInfo      header;
    BSMappingPIExamine          DeviceExamine;
    BSMappingPIInit             Init;
    BSMappingPICleanup          Cleanup;
    BSMappingPIIO               IO;
    BSMappingPIFlush            Flush;
    BSMappingPIAddComponent     AddComponent;
    BSMappingPIGoToState        GotoState;
    BSMappingPIFormatMedia      Format;
    BSMappingPIGetInfo          GetInfo;
    BSMappingIOCompletion       ioCompletion;
};

typedef BSStoreMappingOps *BSStoreMappingOpsPtr;
```

Field descriptions

`header` A `BlockStoragePlugInInfo` structure (page 7-61) that contains the version of the block storage plug-in interface used by this plug-in.

`DeviceExamine` The entrypoint of the mapping plug-in's examine function (page 7-65).

`Init` The entrypoint of the mapping plug-in's initialization function (page 7-66).

`Cleanup` The entrypoint of the mapping plug-in's clean up function (page 7-66).

`IO` The entrypoint of the mapping plug-in's I/O function (page 7-66).

`Flush` The entrypoint of the mapping plug-in's flush function (page 7-67).

AddComponent	The entrypoint of the mapping plug-in's add component function (page 7-68).
GotoState	The entrypoint of the mapping plug-in's accessibility state function (page 7-68).
Format	The entrypoint of the mapping plug-in's format media function (page 7-69).
GetInfo	The entrypoint of the mapping plug-in's information function (page 7-69).
ioCompletion	The entrypoint of the mapping plug-in's I/O completion routine (page 7-67).

BSStorePartitioningOps

A BSStorePartitioningOps structure contains the entrypoints of the functions a partitioning plug-in provides to the block storage family.

A partitioning plug-in must export a structure of type BSStorePartitioningOps named BlockStoragePartitioningPIOps to the block storage family.

```
struct BSStorePartitioningOps {
    BlockStoragePlugInInfo              header;
    BSPartitioningPIExamine             Examine;
    BSPartitioningPIInit                Init;
    BSPartitioningPICleanup             Cleanup;
    BSPartitioningPIInitializeMap       InitializeMap;
    BSPartitioningPIGetInfo             GetInfo;
    BSPartitioningPIGetEntry            GetEntry;
    BSPartitioningPISetEntry            SetEntry;
};

typedef BSStorePartitioningOps *BSStorePartitioningOpsPtr;
```

Field descriptions

header	A BlockStoragePlugInInfo structure (page 7-61) that contains the version of the block storage plug-in interface used by this plug-in.
Examine	The entrypoint of the partitioning plug-in's examine function (page 7-70).

`Init`	The entrypoint of the partitioning plug-in's initialization function (page 7-71).
`Cleanup`	The entrypoint of the partitioning plug-in's clean up function (page 7-71).
`InitializeMap`	The entrypoint of the partitioning plug-in's initialize map function (page 7-72).
`GetInfo`	The entrypoint of the partitioning plug-in's information function (page 7-72).
`GetEntry`	The entrypoint of the partitioning plug-in's get entry function (page 7-72).
`SetEntry`	The entrypoint of the partitioning plug-in's set entry function (page 7-73).

BSContainerPolicyOps

A `BSContainerPolicyOps` structure contains the entrypoints of the functions a container plug-in provides to the block storage family.

A container plug-in must export a structure of type `BSContainerPolicyOps` named `BlockStorageContainerPIOps` to the block storage family.

```
struct BSContainerPolicyOps {
    BlockStoragePlugInInfo              header;
    BSContainerPIExamine                Examine;
    BSContainerPIInit                   Init;
    BSContainerPICleanup                Cleanup;
    BSContainerPIGoToState              GoToState;
    BSContainerPIAddContainer           AddContainer;
    BSContainerPIGetInfo                GetInfo;
    BSContainerPIBackgroundTask         BackgroundTask;
};

typedef BSContainerPolicyOps *BSContainerPolicyOpsPtr;
```

Field descriptions

`header`	A `BlockStoragePlugInInfo` structure (page 7-61) that contains the version of the block storage plug-in interface used by this plug-in.

`Examine`	The entrypoint of the container plug-in's examine function (page 7-74).
`Init`	The entrypoint of the container plug-in's initialization function (page 7-74).
`Cleanup`	The entrypoint of the container plug-in's clean up function (page 7-75).
`GoToState`	The entrypoint of the container plug-in's accessibility state function (page 7-75).
`AddContainer`	The entrypoint of the container plug-in's add container function (page 7-75).
`GetInfo`	The entrypoint of the container plug-in's information function (page 7-76).
`BackgroundTask`	The entrypoint of the container plug-in's initialization background task function (page 7-76).

Mapping Plug-in-Defined Function Types

This section describes the function types that a mapping plug-in exports to the block storage family.

BSMappingPIExamine

Prior to selecting a mapping plug-in to manage a given store, the block storage expert calls the examine function of each mapping plug-in. A mapping plug-in examines the store and then returns an indication of how well it can support it.

A mapping plug-in exports a pointer to its examine function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIExamine)(
                            BSStorePtr examineStore,
                            BSMPIConfidenceLevel *confidence);
```

For information about creating your own examine function, see the description of the `MyBSMappingPIExamineFunc` function (page 7-179).

BSMappingPIInit

When the block storage expert selects a mapping plug-in to manage a given store, it calls the initialization function provided by the plug-in. The plug-in then prepares itself to handle requests to that store.

A mapping plug-in exports a pointer to its initialization function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIInit)(BSStorePtr initStore);
```

For information about creating your own initialization function, see the description of the `MyBSMappingPIInitFunc` function (page 7-181).

BSMappingPICleanup

Before the block storage family deletes a store, it calls the clean up function provided by the mapping plug-in that manages that store. The plug-in then completes processing and release resources associated with the store.

A mapping plug-in exports a pointer to its clean up function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPICleanup)(BSStorePtr cleanupStore);
```

For information about creating your own clean up function, see the description of the `MyBSMappingPICleanupFunc` function (page 7-182).

BSMappingPIIO

When the block storage family gets an I/O request for a given store, it calls the I/O function provided by the mapping plug-in that manages the store. The plug-in then processes the request.

A mapping plug-in exports a pointer to its I/O function. The function pointer is defined by the block storage family as follows:

```
typedef extern BSIOStatus (*BSMappingPIIO)(
                            BSStorePtr ioStore,
                            BSBlockListDescriptorRef blocks,
                            MemListDescriptorRef memory,
                            BSIORequestBlockPtr parentRequest,
                            OptionBits options,
                            BSErrorList **errors);
```

For information about creating your own I/O function, see the description of the `MyBSMappingPIIOFunc` function (page 7-183).

BSMappingIOCompletion

When the block storage family gets a notification that an I/O request inititated by a mapping plug-in has completed, it calls the I/O completion routine provided by the mapping plug-in that initiated the request.

A mapping plug-in exports a pointer to its I/O completion routine. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingIOCompletion)(
                            BSStorePtr theStore,
                            void *finishedPrivateData,
                            BSErrorListPtr returnedBSErrorList,
                            OSStatus returnedStatus,
                            BSErrorListPtr *errorListPtrPtr);
```

For information about creating your own I/O completion routine, see the description of the `MyBSMappingIOCompletionFunc` function (page 7-185).

BSMappingPIFlush

When the block storage family gets a request to flush a store's cache, it calls the flush function provided by the mapping plug-in that manages the store. The plug-in then processes the request.

A mapping plug-in exports a pointer to its flush function. The function pointer is defined by the block storage family as follows:

```
typedef extern BSIOStatus (*BSMappingPIFlush)(
                            BSStorePtr ioStore,
                            BSIORequestBlockPtr parentRequest,
                            BSErrorList **errors);
```

For information about creating your own flush function, see the description of the `MyBSMappingPIFlushFunc` function (page 7-186).

BSMappingPIAddComponent

After the block storage family selects a mapping plug-in to manage a given store and has called the plug-in's initialization function, if a new component is added to the store, the family calls the add component function provided by the plug-in. The plug-in then processes the request.

A mapping plug-in exports a pointer to its add component function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIAddComponent)(
                            BSStorePtr destStore,
                            BSStoreMPIComponent *newComponent,
                            BSStoreInfo *storeNewInfo);
```

For information about creating your own add component function, see the description of the `MyBSMappingPIAddComponentFunc` function (page 7-187).

BSMappingPIGoToState

When the block storage family receives a request to change the accessibility state of a store, it calls the accessibility state function provided by the mapping plug-in that manages the store. The plug-in then processes the request.

A mapping plug-in exports a pointer to its accessibility state function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIGoToState)(
                            BSStorePtr theStore,
                            BSAccessibilityState gotoState);
```

For information about creating your own accessibility state function, see the description of the `MyBSMappingPIGoToStateFunc` function (page 7-188).

BSMappingPIFormatMedia

When the block storage family gets a format request for a given store, it calls the format media function provided by the mapping plug-in associated with that store. The plug-in then processes the request.

A mapping plug-in exports a pointer to its format media function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIFormatMedia)(
                            BSStorePtr formatStore,
                            BSFormatIndex formatType);
```

For information about creating your own format media function, see the description of the `MyBSMappingPIFormatMediaFunc` function (page 7-189).

BSMappingPIGetInfo

When the block storage family gets a request for information about a given store, it calls the information function provided by the mapping plug-in that manages that store. The plug-in then processes the request.

A mapping plug-in exports a pointer to its information function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMappingPIGetInfo)(
                            BSStorePtr infoStore,
                            BSStoreMPIInfo *info);
```

For information about creating your own information function, see the description of the `MyBSMappingPIGetInfoFunc` function (page 7-190).

BSMPIBackgroundTask

When a mapping plug-in calls the `BSMPIStartBackgroundTask` function (page 7-165), it provides a pointer to a background task function. The block storage family responds by calling the plug-in's background task function.

The background task function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSMPIBackgroundTask)(void *theArg);
```

For information about creating your own background task function, see the description of the `MyBSMPIBackgroundTaskFunc` function (page 7-191).

Partitioning Plug-in-Defined Function Types

This section describes the function types that a partitioning plug-in exports to the block storage family.

BSPartitioningPIExamine

Prior to selecting a partitioning plug-in for a given store, the block storage expert calls the examine function of each partitioning plug-in. A partitioning plug-in examines the store and reports if it recognizes the partition map format.

A partitioning plug-in exports a pointer to its examine function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPIExamine)(
                            BSStoreConnID readStoreConn,
                            UInt32 *certainty);
```

For information about creating your own examine function, see the description of the `MyBSPartitioningPIExamineFunc` function (page 7-192).

BSPartitioningPIInit

When the block storage expert selects a partitioning plug-in for a store, it calls the initialization function provided by the plug-in. The partitioning plug-in then prepares itself to handle requests pertaining to that store.

A partitioning plug-in exports a pointer to its initialization function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPIInit)(
                            BSStorePtr initStore);
```

For information about creating your own initialization function, see the description of the `MyBSPartitioningPIInitFunc` function (page 7-193).

BSPartitioningPICleanup

Before the block storage family deletes a given store, it calls the clean up function provided by the partitioning plug-in associated with that store. The plug-in then releases resources associated with the store.

A partitioning plug-in exports a pointer to its clean up function. The function pointer is defined by the block storage family as follows:

```
typedef extern void (*BSPartitioningPICleanup)(
                            BSStorePtr cleanupStore);
```

For information about creating your own clean up function, see the description of the `MyBSPartitioningPICleanupFunc` function (page 7-193).

BSPartitioningPIInitializeMap

When a partition map needs to be initialized, the block storage family calls the initialize map function provided by the partitioning plug-in associated with it.

A partitioning plug-in exports a pointer to its initialize map function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPIInitializeMap)(
                            BSStorePtr initStore);
```

For information about creating your own initialize map function, see the description of the `MyBSPartitioningPIInitializeMapFunc` function (page 7-194).

BSPartitioningPIGetInfo

When the block storage family receives a request for information about a given store, it calls the information function provided by the partitioning plug-in associated with that store. The plug-in then processes the request.

A partitioning plug-in exports a pointer to its information function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPIGetInfo)(
                            BSStorePtr store,
                            BSStorePPIInfo *info);
```

For information about creating your own information function, see the description of the `MyBSPartitioningPIGetInfoFunc` function (page 7-195).

BSPartitioningPIGetEntry

The block storage family calls the get entry function provided by the partitioning plug-in associated with a store. The plug-in then processes the request.

Typically, this function is called during boot time and in response to requests for information from disk setup applications.

A partitioning plug-in exports a pointer to its get entry function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPIGetEntry)(
                            BSStorePtr readStore,
                            ItemCount entryNum,
                            BSPartitionDescriptor *retEntry));
```

For information about creating your own get entry function, see the description of the `MyBSPartitioningPIGetEntryFunc` function (page 7-196).

BSPartitioningPISetEntry

To define a partition, the block storage family calls the set entry function provided by the partitioning plug-in associated with a store. The plug-in then processes the request.

A partitioning plug-in exports a pointer to its set entry function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSPartitioningPISetEntry)(
                            BSStorePtr store,
                            ItemCount partitionNum,
                            BSPartitionDescriptor *partitionInfo));
```

For information about creating your own set entry function, see the description of the `MyBSPartitioningPISetEntryFunc` function (page 7-197).

Container Plug-in-Defined Function Types

This section describes the function types that a container plug-in exports to the block storage family.

BSContainerPIExamine

Prior to selecting a container plug-in for a container, the block storage expert calls the examine function of each container plug-in. A container plug-in examines the container and return an indication of how well it can support it.

A container plug-in exports a pointer to its examine function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIExamine)(
                            BSContainerPtr initContainer,
                            BSCPIConfidenceLevel *levelOfConfidence);
```

For information about creating your own examine function, see the description of the `MyBSContainerPIExamineFunc` function (page 7-198).

BSContainerPIInit

When the block storage expert selects a container plug-in for a container, it calls the initialization function provided by the plug-in. The plug-in then prepares itself to handle requests related to that container.

A container plug-in exports a pointer to its initialization function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIInit)(
                            BSContainerPtr initContainer,
                            BSContainerPIInfo *info,
                            Boolean *backgroundTask);
```

For information about creating your own initialization function, see the description of the `MyBSContainerPIInitFunc` function (page 7-199).

BSContainerPICleanup

Before the block storage family deletes a given container, it calls the clean up function provided by the container plug-in for that container. The plug-in then completes processing and release resources associated with the container.

A container plug-in exports a pointer to its clean up function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPICleanup)(
                            BSContainerPtr container);
```

For information about creating your own clean up function, see the description of the `MyBSContainerPICleanupFunc` function (page 7-200).

BSContainerPIAddContainer

The block storage family calls a container plug-in's add container function to inform it that another container is being inserted into the plug-in's container.

A container plug-in exports a pointer to its add container function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIAddContainer)(
                            BSContainerPtr destContainer,
                            BSContainerPtr addedContainer);
```

For information about creating your own add container function, see the description of the `MyBSContainerPIAddContainerFunc` function (page 7-203).

BSContainerPIGoToState

When the block storage family receives a request to change the accessibility state for a given container, it calls the accessibility state function provided by the container plug-in for that container. The plug-in then processes the request.

A container plug-in exports a pointer to its accessibility state function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIGoToState)(
                            BSContainerPtr container,
                            UInt32 accessState);
```

For information about creating your own accessibility state function, see the description of the `MyBSContainerPIGoToStateFunc` function (page 7-201).

BSContainerPIGetInfo

The block storage family calls the information function provided by a container plug-in to get information about a container. The plug-in then processes the request.

A container plug-in exports a pointer to its information function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIGetInfo)(
                            BSContainerPtr infoContainer,
                            BSContainerPIInfo *info);
```

For information about creating your own information function, see the description of the `MyBSContainerPIGetInfoFunc` function (page 7-202).

BSContainerPIBackgroundTask

Container plug-ins have two types of background tasks that differ in their interface and when they are called.

If the `backgroundTask` flag is set on exit from a container plug-in's initialization function (page 7-199), the block storage family calls the plug-in's initialization background task function at that time.

See "BSCPIBackgroundTask" (page 7-77) for information about the standard type of background task.

A container plug-in exports a pointer to its initialization background task function. The function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSContainerPIBackgroundTask)(
                            BSContainerPtr container);
```

For information about creating your own initialization background task function, see the description of the `MyBSContainerPIBackgroundTaskFunc` function (page 7-203).

BSCPIBackgroundTask

Container plug-ins have two types of background tasks that differ in their interface and when they are called.

When a container plug-in calls the `BSCPIStartBackgroundTask` function (page 7-176), it provides a pointer to a standard background task function, which the block storage family then calls.

See "BSContainerPIBackgroundTask" (page 7-76) for information about the initialization type of background task.

The standard background task function pointer is defined by the block storage family as follows:

```
typedef extern OSStatus (*BSCPIBackgroundTask)(
                            BSContainerPtr theContainer,
                            void *theArg);
```

For information about creating your own standard background task function, see the description of the `MyBSCPIBackgroundTaskFunc` function (page 7-204).

Block Storage Client Functions

This section describes the functions in the programming interface that the block storage family provides for its clients.

Clients who make I/O requests of existing stores use the functions described in

- "Opening and Closing a Connection to a Store" (page 7-78)
- "Building a Block List" (page 7-80)
- "Reading From a Store" (page 7-86)
- "Writing To a Store" (page 7-92)

Such clients might also need the functions described in "Setting the Accessibility State For a Store" (page 7-98) and "Navigating a Store Hierarchy" (page 7-100).

Clients who create and configure new and existing stores need the functions described in "Creating and Configuring a Store" (page 7-113).

Note
Although the functions in "Working With a Block List Descriptor" (page 7-147) are available to block storage clients, they are not typically used by clients. Usually, mapping plug-ins call them to manipulate block list descriptors in servicing an I/O request. ◆

Opening and Closing a Connection to a Store

•••To be provided•••

BSStoreOpen

Opens a store.

```
extern OSStatus BSStoreOpen (
                    BSStoreRef *store,
                    BSStoreOpenOptions options,
                    BSStoreConnID *newConnection);
```

`store` On input, a pointer to a reference to the store that you want to open. You can get a store reference from a new block storage device notification, if you subscribe to such notifications. You can also get a reference from several navigation functions decribed in "Navigating a Store Hierarchy" (page 7-100).

`options` The type of connection you are requesting. See "BSStoreOpenOptions" (page 7-42) for a description of connection types.

`newConnection` A pointer to a connection ID. On output, the function provides an ID for the newly created connection. You use the ID with other functions to read and write to the store and otherwise manipulate the store.

function result A result code. Your request to open a store can fail if resources for the connection cannot be allocated or if the access options you specify are incompatible with the store or with existing connections. If a store has already granted an exclusive connection, all new connection requests fail with the `E_BSStoreInUse` result code. The function also returns `E_BSStoreInUse` if you request an exclusive connection to a store that has another connection open. If you request write access to a read-only store, the function returns the `E_BSStoreWriteProtected` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You cannot read or write to a store until you open it and get a connection ID.

You can open a store after it is published in the name registry.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnClose

Closes a connection to a store.

```
extern OSStatus BSStoreConnClose (BSStoreConnID connection);
```

`connection` The connection ID for the connection you want to close.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Building a Block List

When you request an I/O transfer involving more than one range of bytes on a store, you need to build a block list. A block list specifies the address ranges in the store to be read or written. You use the functions in this section to create an empty block list, add address ranges to it, and signal that the block list is complete. As a result, you get a reference to a block list descriptor which you can pass to read and write functions described in "Reading From a Store" (page 7-86) and "Writing To a Store" (page 7-92).

BSBlockListCreate

Creates an empty block list.

```
extern OSStatus BSBlockListCreate (
                    ItemCount numAnticipatedRanges,
                    BSBlockListRef *newList);
```

`numAnticipatedRanges`
A hint to the block storage family about the number of address ranges you expect to add to the block list. (You can add more ranges than you specify here.)

`newList` A pointer to a block list reference (page 7-31). On output, the function supplies a reference to the new block list.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

If an I/O transfer involves more than one address range, you need a build a block list that contains the starting address and lengths of each range of bytes to be read or written to a store.

After you create an empty block list with the `BSBlockListCreate` function, call the `BSBlockListAddRange` function (page 7-82) to add address ranges to it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSBlockListAddRange

Appends an address range to a block list.

```
extern OSStatus BSBlockListAddRange (
                    BSBlockListRef appendList,
                    BSByteCount startingOffset,
                    BSByteCount length);
```

`appendList` A reference to the block list to which you want to add an address range. You get a block list reference (page 7-31) from the `BSBlockListCreate` function (page 7-81).

`startingOffset` The offset, in bytes, of the first byte of the range you are adding, from the starting address on the store of interest. Store addresses start at 0.

`length` The length, in bytes, of the address range you are adding.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can call the `BSBlockListAddRange` function to add an address range to a block list any time after calling the `BSBlockListCreate` function (page 7-81) to create the block list and before calling the `BSBlockListFinalize` function (page 7-84) to signal its completion. An address range is defined by a starting address and a length. It specifies a location on a store where data will be read or written. You can add as many ranges as are needed for an I/O transfer.

For example, when an application calls the file systems family to read data from a file, the data in the file is typically scattered across the disk in discontiguous chunks. Before calling the `BSStoreConnReadSGAsync` function (page 7-91), the file systems family needs to build a block list that specifies each discontinuous area (range) on the disk that is to be read from.

Assume that the file systems family calls `BSBlockListAddRange` 4 times to add the following ranges:

- 2048 bytes starting at address 8000
- 6144 bytes starting at address 200

- 2560 bytes starting at address 30000
- 5120 bytes starting at address 16000

The total number of bytes to be read is 15,872. Conceptually, what the series of calls to `BSBlockListAddRange` does is it creates 2 linear arrays, each consisting of 15,872 elements. The first array consists of the bytes to be read. It is referred to as the *transfer space*. The second array consists of the addresses of the corresponding bytes in the transfer space.

The order in which you add ranges to a block list is crucial—it determines how the addresses in the transfer space relate to addresses on a device.

To continue the example, the block list information is eventually used to read the data from a device. Suppose the disk has a maximum read block size of 4096 bytes. The disk driver issues the following requests to the device:

- read 2048 bytes starting at address 8000
- read 4096 bytes starting at address 200
- read 2048 bytes starting at address 4296
- read 2560 bytes starting at address 30000
- read 4096 bytes starting at address 16000
- read 1024 bytes starting at address 20096

(The example ignores the fact that typically the addresses provided in the initial I/O are not actual device addresses and need to be mapped to corresponding device addresses.)

When you are done building your block list, you must signal that all address ranges have been added by calling the `BSBlockListFinalize` function (page 7-84).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSBlockListFinalize

Completes a block list and creates a block list descriptor.

```
extern OSStatus BSBlockListFinalize (
                    BSBlockListRef finalizeList,
                    BSBlockListDescriptorRef *newDescriptor);
```

`finalizeList` The block list reference (page 7-31) that identifies the block list of interest. You get a block list reference from the `BSBlockListCreate` function (page 7-81).

`newDescriptor` A pointer to a block list descriptor reference. On output, the function supplies a reference to the newly created block list descriptor.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

After building a block list with the `BSBlockListAddRange` function (page 7-81), you call the `BSBlockListFinalize` function to signal that you are done adding address ranges to the block list.

The `BSBlockListFinalize` function builds a block list descriptor, an opaque structure that specifies a given view of the block list itself, and returns a reference to it. You pass the reference to block storage read and write functions,

described in "Reading From a Store" (page 7-86) and "Writing To a Store" (page 7-92).

After calling `BSBlockListFinalize`, you cannot call `BSBlockListAddRange` again to add another range to the list.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSBlockListDelete

Disposes of a block list.

```
extern OSStatus BSBlockListDelete (BSBlockListRef deleteList);
```

`deleteList` The block list reference (page 7-31) that identifies the block list you want to dispose of.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

Under normal circumstances, you do not need to call the `BSBlockListDelete` function. Rather, you call the `BSBlockListDescriptorDelete` function (page 7-156) to dispose of a block list descriptor and free the associated block list.

Use `BSBlockListDelete` to dispose of a block list in an abnormal situation when proper disposal of block list descriptors cannot be guaranteed. It relinquishes

all resources associated with the block list, including all its block list descriptors.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Reading From a Store

You use the functions in this section to read data from a store. To read data from a single address range, you can call either the `BSStoreConnRead` (page 7-86) or the `BSStoreConnReadAsync` (page 7-88) function. To read data from more than one address range, use the `BSStoreConnReadSG` (page 7-89) or the `BSStoreConnReadSGAsync` (page 7-91) function.

BSStoreConnRead

Reads a single range of data from a store.

```
extern OSStatus BSStoreConnRead (
                    BSStoreConnID readConnection,
                    BSByteCount startingOffset,
                    BSByteCount bytesToRead,
                    void *buffer);
```

`readConnection`

Your connection ID for the store from which you want to read. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`startingOffset`
: The offset, in bytes, from the beginning of the store at which to start reading. Byte numbering within a store always starts at 0.

`bytesToRead`
: The number of bytes that you want to read.

`buffer`
: A pointer to your buffer. On output, the function puts the data from the store into your buffer.

function result
: A result code. If the range request exceeds the boundaries of the store, the function returns the `E_BSMPIOutOfStoreBounds` result code. The function returns `E_BSMPIMemoryAccessFault` if the destination memory is not accessible. If the store maps to media that is not available, the function returns the `E_BSMPIMediaRemoved` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You use the `BSStoreConnRead` function when you need to read data synchronously from a single address range on a store.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To read data asynchronously from a single address range, use the `BSStoreConnReadAsync` function (page 7-88).

To read data from multiple address ranges, use the `BSStoreConnReadSG` (page 7-89) and `BSStoreConnReadSGAsync` (page 7-91) functions.

BSStoreConnReadAsync

Reads a single range of data from a store asynchronously.

```
extern OSStatus BSStoreConnReadAsync (
                    BSStoreConnID readConnection,
                    BSByteCount startingOffset,
                    BSByteCount bytesToRead,
                    KernelNotification *notification,
                    void *buffer);
```

`readConnection`
Your connection ID for the store from which you want to read. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`startingOffset`
The offset, in bytes, from the beginning of the store at which to start reading. Byte numbering within a store always starts at 0.

`bytesToRead` The number of bytes that you want to read.

`notification` On input, a pointer to a kernel notification structure that specifies how you wish to be notified when the read operation completes. For information on kernel notification structures, see *Microkernel and Core System Services*.

`buffer` A pointer to your buffer. On output, the function puts the data it read from the store into your buffer.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To read data synchronously from a single address range, use the `BSStoreConnRead` function (page 7-86).

To read data from multiple address ranges, use the `BSStoreConnReadSG` (page 7-89) and `BSStoreConnReadSGAsync` (page 7-91) functions.

BSStoreConnReadSG

Reads multiple ranges of data from a store synchronously.

```
extern OSStatus BSStoreConnReadSG (
                    BSStoreConnID readConnection,
                    BSBlockListDescriptorRef srcBlocks,
                    MemListDescriptorRef destMemory);
```

`readConnection`
Your connection ID for the store from which you want to read. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`srcBlocks` A reference to a block list descriptor (page 7-32) that specifies the address ranges that you want to read. For information on how to get a block list descriptor, see "Building a Block List" (page 7-80).

`destMemory`	A reference to a memory list descriptor that specifies where in memory to put the data read from the store. For information on memory lists, see "Memory Lists", a chapter to be provided in a later Developer Release.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To read data asynchronously from multiple address ranges, use the `BSStoreConnReadSGAsync` (page 7-91) function.

To read data froma single address range, use the `BSStoreConnRead` (page 7-86) and `BSStoreConnReadAsync` (page 7-88) functions.

BSStoreConnReadSGAsync

Reads multiple ranges of data from a store asynchronously.

```
extern OSStatus BSStoreConnReadSGAsync (
                    BSStoreConnID readConnection,
                    BSBlockListDescriptorRef srcBlocks,
                    MemListDescriptorRef destMemory,
                    KernelNotification *notification);
```

`readConnection`
: Your connection ID for the store from which you want to read. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`srcBlocks`
: A reference to a block list descriptor (page 7-32) that specifies the address ranges that you want to read. For information on how to get a block list descriptor, see "Building a Block List" (page 7-80).

`destMemory`
: A reference to a memory list descriptor that specifies where in memory to put the data read from the store. For information on memory lists, see "Memory Lists", a chapter to be provided in a later Developer Release.

`notification`
: On input, a pointer to kernel notification structure that specifies how you wish to be notified when the read operation completes. For information on kernel notification structures, see *Microkernel and Core System Services*.

function result
: A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You must not deallocate or modify the block list and memory list structures, nor modify the contents of the memory locations specified by the memory list or free that memory, until after the I/O request completes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Writing To a Store

You use the functions in this section to write data to a store. To write data to a single address range, you can call either the `BSStoreConnWrite` (page 7-92) or the `BSStoreConnWriteAsync` (page 7-93) function. To write data to more than one address range, use the `BSStoreConnWriteSG` (page 7-95) or the `BSStoreConnWriteSGAsync` (page 7-96) function.

BSStoreConnWrite

Writes a single range of data to a store.

```
extern OSStatus BSStoreConnWrite (
                    BSStoreConnID writeConnection,
                    BSByteCount startingOffset,
                    ByteCount bytesToWrite,
                    void *buffer);
```

`writeConnection`

Your connection ID for the store to which you want to write. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`startingOffset`

The offset, in bytes, from the beginning of the store at which to start writing. Byte numbering within a store always starts at 0.

`bytesToWrite`	The number of bytes that you want to write.
`buffer`	On input, a pointer to your buffer containing the data to write.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To write data asynchronously to a single address range, use the `BSStoreConnWriteAsync` function (page 7-93).

To write data to multiple address ranges, use the `BSStoreConnWriteSG` (page 7-95) and `BSStoreConnWriteSGAsync` (page 7-96) functions.

BSStoreConnWriteAsync

Writes a single range of data to a store asynchronously.

```
extern OSStatus BSStoreConnWriteAsync (
                    BSStoreConnID writeConnection,
                    BSByteCount startingOffset,
```

```
                    ByteCount bytesToWrite,
                    KernelNotification *notification,
                    void *buffer);
```

`writeConnection`
Your connection ID for the store to which you want to write. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`startingOffset`
The offset, in bytes, from the beginning of the store at which to start writing. Byte numbering within a store always starts at 0.

`bytesToWrite` The number of bytes that you want to write.

`notification` On input, a pointer to kernel notification structure that specifies how you wish to be notified when the write operation completes. For information on kernel notification structures, see *Microkernel and Core System Services*.

`buffer` On input, a pointer to your buffer containing the data to write.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To write data synchronously to a single address range, use the `BSStoreConnWrite` function (page 7-92).

To write data to multiple address ranges, use the `BSStoreConnWriteSG` (page 7-95) and `BSStoreConnWriteSGAsync` (page 7-96) functions.

BSStoreConnWriteSG

Writes multiple ranges of data to a store synchronously.

```
extern OSStatus BSStoreConnWriteSG (
                    BSStoreConnID writeConnection,
                    MemListDescriptorRef srcMemory,
                    BSBlockListDescriptorRef destBlocks);
```

`writeConnection`
Your connection ID for the store to which you want to write. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`srcMemory` A reference to a memory list descriptor that tells the function where in memory to find the data that you want to write. For information on memory lists, see "Memory Lists", a chapter to be provided in a later Developer Release.

`destBlocks` A reference to a block list descriptor that specifies the address ranges on the store where you want to write the data. See "Building a Block List" (page 7-80) for information on getting a block list descriptor.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To write data asynchronously to multiple address ranges, use the `BSStoreConnWriteSGAsync` function (page 7-96).

To write data to a single address range, use the `BSStoreConnWrite` (page 7-92) and `BSStoreConnWriteAsync` (page 7-93) functions.

BSStoreConnWriteSGAsync

Writes multiple ranges of data to a store asynchronously.

```
extern OSStatus BSStoreConnwriteSGAsync (
                    BSStoreConnID writeConnection,
                    MemListDescriptorRef srcMemory,
                    BSBlockListDescriptorRef destBlocks,
                    KernelNotification *notification);
```

`writeConnection`
Your connection ID for the store to which you want to write. You get a connection ID from the `BSStoreOpen` function (page 7-78).

`srcMemory` A reference to a memory list descriptor that tells the function where in memory to find the data that you want to write. For information on memory lists, see “Memory Lists”, a chapter to be provided in a later Developer Release.

`destBlocks`	A reference to a block list descriptor that specifies the address ranges on the store where you want to write the data. See "Building a Block List" (page 7-80) for information on getting a block list descriptor.
`notification`	On input, a pointer to kernel notification structure that specifies how you wish to be notified when the write operation completes. For information on kernel notification structures, see *Microkernel and Core System Services*.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You must not deallocate or modify the block list and memory list structures, nor modify the contents of the memory locations specified by the memory list or free that memory, until after the I/O request completes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To write data synchronously to multiple address ranges, use the `BSStoreConnWriteSG` function (page 7-95).

To write data to a single address range, use the `BSStoreConnWrite` (page 7-92) and `BSStoreConnWriteAsync` (page 7-93) functions.

BSStoreConnFlush

Flushes caches in a store and any stores from which the store is derived.

```
extern OSStatus BSStoreConnFlush (BSStoreConnID flushConnection);
```

`flushConnection`
Your connection ID for the store that you want to flush. You get a connection ID from the `BSStoreOpen` function (page 7-78).

function result A result code.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Setting the Accessibility State For a Store

•••To be provided•••

BSStoreConnGoToAccessibilityState

Sets the accessibility state for a store.

```
extern OSStatus BSStoreConnGoToAccessibilityState (
                    BSStoreConnID connection,
                    BSAccessibilityState newState);
```

`connection` The connection ID for the store whose state you want to set.

`newState` The new accessibility state that you want to set. See "Accessibility State Type" (page 7-41) for descriptions of the defined states.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

If the new state is `kBSOffline` and the store maps to a device or media that is ejectable, the block storage family deletes the store from the store hierarchy and closes the connection.

Furthermore, if the store of interest is a leaf store and none of the stores to which it maps is in use, the action is repeated through the related stores so that an entire branch in the store hierarchy is deleted.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Navigating a Store Hierarchy

The functions in this section allow you to browse a hierarchy of stores and retrieve properties of a given store. They are useful primarily to disk configuration utility software.

See "Navigation Types" (page 7-33) for information on the data types defined for store navigation.

You do not need a store connection ID to use the navigation functions.

You can get a store reference, which you need to open a store and get a connection to it, by calling any of these functions: `BSStoreIteratorEnter`, `BSStoreIteratorExit`, `BSStoreIteratorRestartChildren`, `BSStoreIteratorRestartParent`, `BSStoreIteratorNextChild` and `BSStoreIteratorNextParent`. If you know the ID of a store, the `BSStoreFindByID` function also returns a store reference.

BSStoreGetDeviceData

Retrieves information about all stores, leaf stores, or primary stores.

```
extern OSStatus BSStoreGetDeviceData(
                    BSStoreGetSelector selector,
                    ItemCount requestItemCount,
                    ItemCount *totalItemCount,
                    BSStoreIOIteratorData *iteratorData);
```

`selector` A value that indicates the set of stores about which you want information. "BSStoreGetSelector" (page 7-33) contains descriptions of the values you can use here.

`requestItemCount`
The number of empty `BSStoreIOIteratorData` structures in the array pointed to by the `iteratorData` parameter.

`totalItemCount`
A pointer to an `ItemCount` value. On output, the function sets this parameter to the number of stores of the type you specified

for which information is available. This value can be more than the number of `BSStoreIOIteratorData` structures in your array. In that case, the function fills in every structure in the array.

`iteratorData` On input, a pointer to an array of empty `BSStoreIOIteratorData` structures (page 7-34). On output, the function fills in the fields of a structure for each store it finds or until it reaches the end of the array.

function result A result code.

DISCUSSION

Although the `BSStoreGetDeviceData` function name refers to a device (due to I/O family naming conventions), the function returns information about stores.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorCreate

Creates an iterator that you can use to navigate a store hierarchy.

```
extern OSStatus BSStoreIteratorCreate (
                    BSStoreRef *startingStore,
                    BSStoreIteratorID *newIterator);
```

`startingStore` On input, a pointer to the store reference for the new iterator's starting store (the store at which it will begin an iteration). If you set this parameter to `nil`, the iterator points at the root of the block storage store hierarchy.

`newIterator` A pointer to an iterator (page 7-34). On output, the function supplies a new iterator.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

If you set the `startingStore` parameter to `nil` to start at the root of the store hierarchy, you can call the `BSStoreIteratorNextChild` function (page 7-107) repeatedly to discover the child nodes of the root.

When you are done browsing the hierarchy, call the `BSStoreIteratorDispose` function (page 7-102) to dispose of the iterator and any resources associated with it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorDispose

Disposes of an iterator, freeing all resources associated with it.

```
extern OSStatus BSStoreIteratorDispose (
                    BSStoreIteratorID disposeIterator);
```

`disposeIterator`
The iterator you want to dispose of.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You need to call this function for each iterator that you create with the `BSStoreIteratorCreate` function (page 7-101).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorEnter

Updates an iterator to point to the first child of the current node and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorEnter (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newStore);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the current node's first child. If the function returns an error, the iterator is unchanged.

`newStore` A pointer to a store reference. On output, the function sets the reference to the store the updated iterator points to. If no child node exists, the function sets the pointer to `nil`.

function result A result code. If no child node exists, the function returns the `E_BSStoreNotFound` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

When you call the `BSStoreIteratorEnter` function, the iterator points to the current node. As a result of successful execution, the iterator enters a new level of the hierarchy—it moves down a level and points to the first child of the node it previously pointed to.

You can use the function to begin an iteration through the children of a node.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorExit

Updates an iterator to point to the first parent of the current node and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorExit (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newStore);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the current node's first parent. If the function returns an error, the iterator is unchanged.

`newStore`	A pointer to a store reference. On output, the function sets the reference to the store the updated iterator points to. If no parent node exists, the function sets the pointer to `nil`.
function result	A result code. If no parent node exists, the function returns the `E_BSStoreNotFound` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

When you call the `BSStoreIteratorExit` function, the iterator points to the current node. As a result of successful execution, the iterator enters a new level of the hierarchy—it moves up a level and points to the first parent of the node it previously pointed to.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorRestartChildren

Updates an iterator to point to the first child in the set of child nodes it is currently in and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorRestartChildren (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newChild);
```

`iterator`	An iterator that you provide. The function updates the iterator to point to the first child in the current set of sibling nodes.
`newChild`	A pointer to a store reference. On output, the function sets the reference to the store the updated iterator points to.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The child nodes of a given node constitute a set of sibling nodes. The `BSStoreIteratorRestartChildren` function sets an iterator to the first child in the sibling set.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorRestartParent

Updates an iterator to point to the first parent in the current set and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorRestartParent (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newParent);
```

`iterator`	An iterator that you provide. The function updates the iterator to point to the first parent in the current set of parent nodes.

`newParent`	A pointer to a store reference. On output, the function sets the reference to the store the updated iterator points to.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The parent nodes of a given node constitute a set of nodes. The `BSStoreIteratorRestartParent` function sets an iterator to the first node in the current set of parent nodes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorNextChild

Updates an iterator to point to the next child in the current sibling set and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorNextChild (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newChild,
                    Boolean *changed);
```

`iterator`	An iterator that you provide. The function updates the iterator to point to the next child in the current sibling set.
`newChild`	A pointer to a store reference. On output, the function sets the reference to refer to the store the updated iterator points to.

`changed`	A pointer to a Boolean variable. On output, the function sets the variable to `true` if a parent or a child was added to or deleted from the node the iterator pointed to before the function updated it. See the Discussion for more information.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can use the `BSStoreIteratorNextChild` function move through all the children in the current sibling set.

The block storage family tracks changes in the store hierarchy and uses the `changed` parameter to notify you of some changes. Specifically, it reports a change if a node was added or deleted since the last time either the `BSStoreIteratorNextChild` or `BSStoreIteratorNextParent` (page 7-109) function was called using the same iterator. However, it reports local changes only, not a change anywhere in the hierarchy. That is, it reports changes to the node that was current when you called the function. As a result, you know if the part of the hierarchy is changing, more or less simultaneously, as you examine it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreIteratorNextParent

Updates an iterator to point to the next parent in the current set of nodes and retrieves a reference to the store it represents.

```
extern OSStatus BSStoreIteratorNextParent (
                    BSStoreIteratorID iterator,
                    BSStoreRef *newParent,
                    Boolean *changed);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the next parent in the current set.

`newParent` A pointer to a store reference. On output, the function sets the reference to the store the updated iterator points to.

`changed` A pointer to a Boolean variable. On output, the function sets the variable to `true` if a parent or a child was added to or deleted from the node the iterator pointed to before the function updated it. See the Discussion for more information.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can use the `BSStoreIteratorNextParent` function move through all the nodes in the current set.

The block storage family tracks changes in the store hierarchy and uses the `changed` parameter to notify you of some changes. Specifically, it reports a change if a node was added or deleted since the last time either the `BSStoreIteratorNextChild` (page 7-107) or `BSStoreIteratorNextParent` function was called using the same iterator. However, it reports local changes only, not a change anywhere in the hierarchy. That is, it reports changes to the node that was current when you called the function. As a result, you know if the part of the hierarchy is changing, more or less simultaneously, as you examine it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreGetPropertySize

Retrieves the size of a store property.

```
extern OSStatus BSStoreGetPropertySize (
                    BSStoreRef *store,
                    char *propertyName,
                    BSStorePropertyInstance propertyInstance,
                    ByteCount *propertySize);
```

`store` On input, a pointer to the store reference for the store of interest.

`propertyName` On input, a pointer to a C string containing the name of the property of interest. See "Store Property Names" (page 7-36) for a list of property names.

`propertyInstance`
The instance of the named property of interest. Property instances start at 0 and increment by 1 for each additional instance. You provide this value.

`propertySize` A pointer to a 32-bit value. On output, the function returns the size, in bytes, of the property specified by the `propertyName` and `propertyInstance` parameters.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can call the `BSStoreGetPropertySize` function before calling the `BSStoreGetProperty` function (page 7-111) to find out how large a buffer you should provide to `BSStoreGetProperty`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreGetProperty

Retrieves the value of a store property.

```
extern OSStatus BSStoreGetProperty (
                    BSStoreRef *store,
                    char *propertyName,
                    BSStorePropertyInstance propertyInstance,
                    void *propertyValue,
                    ByteCount *propertySize);
```

`store` On input, a pointer to the store reference for the store of interest.

`propertyName` On input, a pointer to a C string containing the name of the property of interest. See "Store Property Names" (page 7-36) for a list of property names.

`propertyInstance`
The instance of the named property of interest. Property instances start at 0 and increment by 1 for each additional instance. You provide this value.

`propertyValue`	A pointer to your buffer. On output, the function places the property value in your buffer.
`propertySize`	On input, a pointer to a value specifying the size of your buffer in bytes. You can call the `BSStoreGetPropertySize` function (page 7-110) to find out how large a buffer you should provide. On output, the function returns the number of bytes of data it placed in your buffer.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreFindByID

Given the ID of a store, returns a reference to it.

```
extern OSStatus BSStoreFindByID (
                    BSStoreID storeID,
                    BSStoreRef *foundStore);
```

`storeID`	The ID of the store of interest. You get an ID when you create a store (`BSStoreCreate` (page 7-114)). You can also get an ID by locating the store in the store hierarchy and calling the `BSStoreGetProperty` function (page 7-111) to retrieve the ID property.
`foundStore`	A pointer to a store reference. On output, the function returns a reference to the store of interest.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating and Configuring a Store

You use the functions in this section to create a new store, configure it, and make it available for I/O. If you write disk utilities or disk formatting software, you need to understand how to use these functions. If you simply want to use a store for I/O, you do not use these functions.

BSStoreCreate

Creates a new store and opens an exclusive connection to it.

```
extern OSStatus BSStoreCreate (
                    BSStoreID *newStore,
                    BSStoreConnID *newStoreConnection);
```

`newStore` A pointer to a store ID. On output, the function returns the store ID for the new store.

`newStoreConnection`
A pointer to a connection ID. On output, the function provides an ID for a connection to the newly created store. You use the ID with other functions to configure the store. The new connection has exclusive access to the store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

After creating a store, you configure it and make it available for use by other clients by calling the following functions:

- `BSStoreConnAssociatePartitioningPlugin` (page 7-117) and `BSStoreConnAssociateMappingPlugin` (page 7-116) to associate a partitioning and a mapping plug-in with the store
- `BSStoreConnMapDevice` (page 7-122) or `BSStoreConnMapPartition` (page 7-120) to map the store to a device or another store
- `BSStoreConnPublish` (page 7-126) to publish the store inthe name registry
- `BSStoreConnClose` (page 7-80) to close your connection to the store. No other connection to the store can be granted until you close yours.

A client using stores for I/O does not need to call the `BSStoreCreate` function. This function is called by disk utility and disk formatting software.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnDeleteAndClose

Deletes a store.

```
extern OSStatus BSStoreConnDeleteAndClose (BSStoreConnID connection);
```

`connection` The connection ID for the store you want to delete.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can delete a store when the following conditions are met:

- the store is not published in the name registry. (To remove a store from the name registry, call the `BSStoreConnUnpublish` function (page 7-127)).
- there are no stores existing that map to the store to be deleted. (To find out, call `BSStoreConnGetInfo` (page 7-123) and check the `numChildren` field in the returned store information structure.)
- the store has no connections to it other than the one you use when calling the `BSStoreConnDeleteAndClose` function. That connection is closed as a result of successful execution. If another connection exists, the function returns the `E_BSStoreInUse` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnAssociateMappingPlugin

Associates a mapping plug-in with a store.

```
extern OSStatus BSStoreConnAssociateMappingPlugin (
                    BSStoreConnID connection,
                    BSMappingPlugInRef mappingPlugin);
```

`connection` The connection ID for the store.

`mappingPlugin` A reference to the mapping plug-in you want to attach.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You use this function to override automatic plug-in selection. The function returns the `E_BSEPlugInNotFound` result code if the plug-in can't be found.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnAssociatePartitioningPlugin

Associates a partitioning plug-in with a store.

```
extern OSStatus BSStoreConnAssociatePartitioningPlugin (
                    BSStoreConnID connection,
                    BSPartitioningPlugInRef partitioningPlugin);
```

`connection` A connection ID for the store.

`partitioningPlugin`
A reference to the plug-in you want to associate with the store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You use this function to override automatic plug-in selection. The function returns the `E_BSEPlugInNotFound` result code if the plug-in can't be found.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnSetPartitionInfo

Creates or modifies a partition in a store.

```
extern OSStatus BSStoreConnSetPartitionInfo (
                    BSStoreConnID storeConnection,
                    ItemCount partitionNum,
                    BSPartitionDescriptor *partitionInfo);
```

`storeConnection`
The connection ID for the store that does or will contain the partition of interest.

`partitionNum` The ordinal number of the partition you want to create or modify. Partition numbering starts at 0.

`partitionInfo` On input, a pointer to a `BSPartitionDescriptor` structure (page 7-48) containing the partition information you want to set.

function result A result code. If the store does not have a partitioning plug-in, the function returns the `E_BSPPINoPlugIn` result code. If the `BSPartitionDescriptor` structure you provide describes a partition that overlaps an existing partition, the function returns the `E_BSPPIOverlappingPartition` result code. If the new partition extends beyond the boundaries of the store, the function returns the `E_BSPPIOutOfStoreBounds` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You typically call the `BSStoreConnSetPartitionInfo` function to create a new partition in an existing store. The function doesn't automatically create a new store whose limits are defined by the new partition—it simply creates a partition map entry in the existing store.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnGetPartitionInfo

Retrieves information about a partition.

```
extern OSStatus BSStoreConnGetPartitionInfo (
                    BSStoreConnID connection,
                    ItemCount partitionNum,
                    BSPartitionDescriptor *partitionInfo);
```

`connection` The connection ID for the store containing the partition of interest.

`partitionNum` The ordinal number that identifies the partition about which you want information. Partition numbering starts at 0.

`partitionInfo` A pointer to a `BSPartitionDescriptor` structure (page 7-48). On output, the function returns partition information in the structure.

function result A result code. If the store does not have a partitioning plug-in, the function returns the `E_BSPPINoPlugIn` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnMapPartition

Maps a partition from a source store into a destination store.

```
extern OSStatus BSStoreConnMapPartition (
                    BSStoreConnID srcConnection,
                    ItemCount partitionNum,
                    BSStoreConnID destConnection,
                    BSByteCount startingByte);
```

`srcConnection` The connection ID for the source store containing the partition to be mapped.

`partitionNum` The ordinal number, in the source store, of the partition to be mapped.

`destConnection`
The connection ID for the destination store into which the partition is to be mapped.

`startingByte`
The byte offset in the destination store where you want to place the partition.

function result A result code. If the source store does not have a partitioning plug-in, the function returns the `E_BSPPINoPlugIn` result code. If the mapping is not supported by either store, the function returns the `E_BSPPIMappingNotSupported` result code. The function returns `E_BSPPIPartitionNonExistant` if the source partition doesn't exist. If your request would create too many levels in the store hierarchy, the function returns the `E_BSEHierarchyTooDeep` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

Normally, when Mac OS 8 boots, the store hierarchy is built automatically. The `BSStoreConnMapPartition` function offers a programmatic way to add stores to the hierarchy by associating a newly created store with a store already in the hierarchy. By definition, all stores added by `BSStoreConnMapPartition` become derived stores.

You need to call this function if you develop disk utility software. Suppose, for example, that a user adds a uninitialized disk to a Mac-compatible system and wants to split it into 3 virtual disks. You should perform the following actions.

4. Call `BSStoreCreate` (page 7-114) to create a new store, call `BSStoreConnAssociateMappingPlugin` (page 7-116) to associate a mapping plug-in with it, call `BSStoreConnMapDevice` (page 7-122) to map the new disk into the store, and call `BSStoreConnAssociatePartitioningPlugin` (page 7-117) to associate a partitioning plug-in with the store. At this point, you've added a new primary store to the hierarchy.
5. Call `BSStoreConnSetPartitionInfo` (page 7-118) to create the first partition in the primary store.
6. Create a new store, associate a mapping plug-in with it, and call `BSStoreConnMapPartition` to map the first partition from the primary store

into the new store. As a result, the new store is added to the store hierarchy as a derived store.

7. Repeat steps 2 and 3 for the second and third partitions.

The hierarchy with one new primary store and three new derived stores is automatically rebuilt when Mac OS 8 is rebooted.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnMapDevice

Maps a device into a store.

```
extern OSStatus BSStoreConnMapDevice (
                    RegEntryRef srcDevice,
                    BSStoreConnID destConnection);
```

`srcDevice` The name registry reference to the device of interest.

`destConnection`
The connection ID for the destination store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You call the `BSStoreConnMapDevice` function if you need to configure a primary store. (Typically, primary stores are automatically configured by the system.)

After the store is created and a mapping plug-in associated with it, you call `BSStoreConnMapDevice` to map the device into the store. The mapping plug-in must know how to interact with the family that controls the device and do whatever is necessary to map the device.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnGetInfo

Gets information about a store.

```
extern OSStatus BSStoreConnGetInfo (
                    BSStoreConnID infoConnection,
                    BSStoreInfo *infoBuffer);
```

`infoConnection`
The connection ID for the store about which you want information.

`infoBuffer` A pointer to a `BSStoreInfo` structure (page 7-44). On output, the function uses the structure to return information about the store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnGetComponents

Gets the components of a store.

```
extern OSStatus BSStoreConnGetComponents (
                    BSStoreConnID connection,
                    ItemCount tableSize,
                    BSStoreComponent *componentInfo);
```

`connection` The connection ID for the store of interest.

`tableSize` The number of elements in your array pointed to by the `componentInfo` parameter.

`componentInfo` A pointer to your array of `BSStoreComponentInfo` structures (page 7-40). On output, the function uses the structures to supply information about the components of a store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnFormat

Formats a store using a format that you specify.

```
extern OSStatus BSStoreConnFormat (
                    BSStoreConnID connection,
                    BSFormatIndex formatType);
```

`connection` The connection ID for the store that you want to format.

`formatType` The indentifier of the format type you want to use. You can get a format identifier by calling the `BSStoreConnGetInfo` function (page 7-123). That function returns a `BSStoreInfo` structure (page 7-44) from whose `possibleFormats` array you can get a format identifier.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnPublish

Makes a store available for general use.

```
extern OSStatus BSStoreConnPublish (BSStoreConnID connection);
```

`connection` A connection ID for a store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSStoreConnPublish` function adds an entry for a store to the block storage hierarchy in the name registry if the store has a mapping plug-in associated with it and a size greater than 0.

If the store's parent stores have not been published, the function also publishes them if they meet the preceding criteria.

If any store in the hierarchy cannot be published, none of the stores is published.

As a result of successful execution, a new store device notification message is sent to all interested parties.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreConnUnpublish

Removes a store from general use.

```
extern OSStatus BSStoreConnUnpublish (BSStoreConnID connection);
```

`connection` A connection ID for a store.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSStoreConnUnpublish` function removes a store from the block storage hierarchy in the name registry.

You can remove a store if the store does not have any connections to it other than the one you use when calling `BSStoreConnUnpublish`. If another connection exists, the function returns the `E_BSStoreInUse` result code.

The `BSStoreConnUnpublish` function does not delete the store. To do that, call the `BSStoreConnDeleteAndClose` function (page 7-115) after `BSStoreConnUnpublish` returns successfully.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Opening and Closing a Connection to a Container

•••To be provided•••

BSContainerOpen

Opens a container.

```
extern OSStatus BSContainerOpen (
                    BSContainerRef *container,
                    ContainerOpenOptions options,
                    BSContainerConnID *newConnection);
```

`container` On input, a pointer to the container reference for the container you want to open.

`options` The type of access you are requesting for the connection. See "BSContainerOpenOptions" (page 7-43) for a description of the options.

`newConnection` A pointer to a connection ID. On output, the function returns an ID for the newly created connection. You use the ID with other functions to configure the container and change its accessibility state.

function result A result code. Your request to open a container can fail if resources for the connection cannot be allocated or if the options you specify are incompatible with existing connections.

If a container has already granted an exclusive connection, all new connection requests fail with the `E_BSContainerInUse` result code. The function also returns `E_BSContainerInUse` if you request an exclusive connection to a container that has another connection open. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can open a container only after it is published in the name registry.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnClose

Closes a connection to a container.

```
extern OSStatus BSContainerConnClose (BSContainerConnID connection);
```

`connection` The connection ID for the connection you want to close.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Setting the Accessibility State For a Container

•••To be provided•••

BSContainerConnGoToAccessibilityState

Sets the accessibility state for a container.

```
extern OSStatus BSStoreConnGoToAccessibilityState (
                    BSContainerConnID connection,
                    BSAccessibilityState newState);
```

`connection` The connection ID for the container whose state you want to set.

`newState` The new accessibility state that you want to set. See "Accessibility State Type" (page 7-41) for descriptions of the defined states.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Navigating a Container Hierarchy

The functions in this section allow you to browse a hierarchy of containers and retrieve properties of a given container.

BSContainerIteratorCreate

Creates an iterator that you can use to navigate a container hierarchy.

```
extern OSStatus BSContainerIteratorCreate (
                    BSContainerRef *startingContainer,
                    BSContainerIteratorID *newIterator);
```

`startingContainer`
On input, a pointer to the container reference for the new iterator's starting container (the container at which it will begin an iteration). If you set this parameter to `nil`, the iterator points at the root of the block storage container hierarchy.

`newIterator` A pointer to a container iterator (page 7-35). On output, the function supplies a new iterator.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

If you set the `startingContainer` parameter to `nil` to start at the root of the container hierarchy, you can call the `BSContainerIteratorNextChild` function (page 7-136) repeatedly to discover the child nodes of the root.

When you are done browsing the hierarchy, call the `BSContainerIteratorDispose` function (page 7-132) to dispose of the iterator and any resources associated with it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerIteratorDispose

Disposes of an iterator, freeing all resources associated with it.

```
extern OSStatus BSContainerIteratorDispose (
                    BSContainerIteratorID disposeIterator);
```

`disposeIterator`
The iterator you want to dispose of.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You need to call this function for each iterator that you create with the `BSContainerIteratorCreate` function (page 7-131).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerIteratorEnter

Updates an iterator to point to the first child of the current node and retrieves a reference to the container it represents.

```
extern OSStatus BSContainerIteratorEnter (
                    BSContainerIteratorID iterator,
                    BSContainerRef *newContainer);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the current node's first child. If the function returns an error, the iterator is unchanged.

`newContainer` A pointer to a container reference. On output, the function sets the reference to the container the updated iterator points to. If no child node exists, the function sets the pointer to `nil`.

function result A result code. If no child node exists, the function returns an error. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

When you call the `BSContainerIteratorEnter` function, the iterator points to the current node. As a result of successful execution, the iterator enters a new level of the hierarchy—it moves down a level and points to the first child of the node it previously pointed to.

You can use the function to begin an iteration through the children of a node.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerIteratorExit

Updates an iterator to point to the parent of the current node and retrieves a reference to the container it represents.

```
extern OSStatus BSContainerIteratorExit (
                    BSContainerIteratorID iterator,
                    BSContainerRef *newContainer);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the current node's parent. If the function returns an error, the iterator is unchanged.

`newContainer` A pointer to a container reference. On output, the function sets the reference to the container the updated iterator points to. If the parent node does not exist, the function sets the pointer to `nil`.

function result A result code. If the parent node does not exist, the function returns an error. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

When you call the `BSContainerIteratorExit` function, the iterator points to the current node. As a result of successful execution, the iterator enters a new level of the hierarchy—it moves up a level and points to the parent of the node it previously pointed to.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerIteratorRestartChildren

Updates an iterator to point to the first child in the set of child nodes it is currently in and retrieves a reference to the container it represents.

```
extern OSStatus BSContainerIteratorRestartChildren (
                    BSContainerIteratorID iterator,
                    BSContainerRef *newChild);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the first child in the current set of sibling nodes.

`newChild` A pointer to a container reference. On output, the function sets the reference to the container the updated iterator points to.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The child nodes of a given node constitute a set of sibling nodes. The `BSContainerIteratorRestartChildren` function sets an iterator to the first child in the current sibling set.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerIteratorNextChild

Updates an iterator to point to the next child in the current sibling set and retrieves a reference to the container it represents.

```
extern OSStatus BSContainerIteratorNextChild (
                    BSContainerIteratorID iterator,
                    BSContainerRef *newChild);
```

`iterator` An iterator that you provide. The function updates the iterator to point to the next child in the current sibling set.

`newChild` A pointer to a container reference. On output, the function sets the reference to refer to the container the updated iterator points to.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can use the `BSContainerIteratorNextChild` function move through all the children in the current sibling set.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerGetPropertySize

Retrieves the size of a container property.

```
extern OSStatus BSContainerGetPropertySize (
                    BSContainerRef *container,
                    char *propertyName,
                    BSContainerPropertyInstance propertyInstance,
                    ByteCount *propertySize);
```

`container` On input, a pointer to the container reference for the container of interest.

`propertyName` On input, a pointer to a C string containing the name of the property of interest. See "Container Property Names" (page 7-36) for a list of property names.

`propertyInstance`
The instance of the named property of interest. Property instances start at 0 and increment by 1 for each additional instance.

`propertySize` A pointer to a 32-bit value. On output, the function returns the size, in bytes, of the property specified in the `propertyName` and `propertyInstance` parameters.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can call the `BSContainerGetPropertySize` function before calling the `BSContainerGetProperty` function (page 7-138) to find out how large a buffer you should provide to `BSContainerGetProperty`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerGetProperty

Retrieves the value of a container property.

```
extern OSStatus BSContainerGetProperty (
                    BSContainerRef *container,
                    char *propertyName,
                    BSContainerPropertyInstance propertyInstance,
                    void *propertyValue,
                    ByteCount *propertySize);
```

`container` On input, a pointer to the container reference for the container of interest.

`propertyName` On input, a pointer to a C string containing the name of the property of interest. See "Store Property Names" (page 7-36) for a list of property names.

`propertyInstance` The instance of the named property of interest. Property instances start at 0 and increment by 1 for each additional instance.

`propertyValue` A pointer to your buffer. On output, the function places the property value in your buffer.

`propertySize` On input, a pointer to a value specifying the size of your buffer. You can call the `BSContainerGetPropertySize` function (page 7-137) to find out how large a buffer you should provide. On output, the function returns the number of bytes of data it placed in your buffer.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating and Configuring a Container

You use the functions in this section to create and configure a new container.

BSContainerCreate

Creates a new container and opens a connection to it.

```
extern OSStatus BSContainerCreate (BSContainerConnID *newContainer);
```

`newContainer` A pointer to a connection ID. On output, the function provides an ID for a connection to the newly created container. You use the ID with other functions to configure the container. The new connection has exclusive access to the container.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

After creating a container, you need to configure it and make it available for use by other clients by calling the following functions:

- `BSContainerConnAssociatePlugin` (page 7-144) to associate a container plug-in with the container
- `BSContainerConnPublish` (page 7-145) to publish the container in the name registry
- `BSContainerConnClose` (page 7-129) to close your connection to the container

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnDeleteAndClose

Deletes a container.

```
extern OSStatus BSContainerConnDeleteAndClose (
                    BSContainerConnID deleteContainer);
```

`deleteContainer` The connection ID for the container you want to delete.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You can delete a container when the following conditions are met:

- the container is not published in the name registry. (To remove a container from the name registry, call the `BSContainerConnUnpublish` function (page 7-146)).
- there are no containers existing that map to the container to be deleted. (To find out, call `BSContainerConnGetInfo` (page 7-142) and check the `numChildren` field in the returned container information structure.)
- the container has no connections to it other than the one you use when calling the `BSContainerConnDeleteAndClose` function. That connection is closed as a result of successful execution. If another connection exists, the function returns the `E_BSContainerInUse` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnGetInfo

Gets information about a container.

```
extern OSStatus BSContainerConnGetInfo (
                    BSContainerConnID infoConnection,
                    BSContainerInfo *infoBuffer);
```

`infoConnection`
The connection ID for the container of interest.

`infoBuffer` A pointer to a `BSContainerInfo` structure (page 7-47). On output, the function uses the structure to return information about the container.

function result A result code.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnInsertContainer

Inserts a container into another container.

```
extern OSStatus BSContainerConnInsertContainer (
                    BSContainerConnID destContainer,
                    BSContainerConnID putContainer);
```

`destContainer` The connection ID for the container into which you want to insert another container.

`putContainer` The connection ID for the container to be inserted.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnSetDevice

Specifies the physical device a container represents.

```
extern OSStatus BSContainerConnSetDevice (
                    BSContainerConnID connection,
                    RegEntryRef deviceNode);
```

`connection`	The connection ID for the container of interest.
`deviceNode`	The name registry reference for the device that this container represents. You provide the reference.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

If a container does not yet have a device associated with it, you should call this function before calling the `BSContainerConnAssociatePlugIn` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnAssociatePlugIn

Associates a container plug-in with a container.

```
extern OSStatus BSContainerConnAssociatePlugIn (
                    BSContainerConnID connection,
                    BSContainerPlugInRef plugIn);
```

`connection`	The connection ID for the container.
`plugIn`	A reference to the container plug-in you want to attach.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You use this function to override automatic plug-in selection. The function returns the `E_BSEPlugInNotFound` result code if the plug-in can't be found.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnPublish

Makes a container available for general use.

```
extern OSStatus BSContainerConnPublish (
                    BSContainerConnID publishContainer);
```

`publishContainer`
The connection ID for the container of interest.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSContainerConnPublish` function adds an entry for a container to the block storage hierarchy in the name registry if the container has a container plug-in associated with it .

If the container's parent container has not been published, the function publishes it if it meets the preceding criteria.

If any container in the hierarchy cannot be published, none of the containers is published.

As a result of successful execution, a notification is sent to all interested parties.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSContainerConnUnpublish

Removes a container from general use.

```
extern OSStatus BSContainerConnUnpublish (
                    BSContainerConnID unpublishContainer);
```

`unpublishContainer`
The connection ID for the container.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSContainerConnUnpublish` function removes a container from the block storage hierarchy in the name registry.

You can remove a container if the container does not have any connections to it other than the one you use when calling `BSContainerConnUnpublish` and if the container does not have any published children. If another connection exists, the function returns an error.

The `BSContainerConnUnpublish` function does not delete the container. To do that, call the `BSContainerConnDeleteAndClose` function (page 7-141) after `BSContainerConnUnpublish` returns successfully.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Working With a Block List Descriptor

The functions in this section allow you to manipulate block list descriptors. Although these functions are available to block storage clients, a client does not typically use them.

However, mapping plug-ins usually need to manipulate block list descriptors and as a result, they are the primary users of the functions described here.

BSBlockListDescriptorGetInfo

Returns information about a block list descriptor.

```
extern OSStatus BSBlockListDescriptorGetInfo (
                    BSBlockListDescriptorRef infoDescriptor,
                    BSBlockListDescriptorInfo *info);
```

`infoDescriptor`
A reference to the block list descriptor (page 7-32) about which you want information.

`info`	A pointer to a `BSBlockListDescriptorGetInfo` structure (page 7-51). On output, the function returns information about the block list descriptor in the structure.
function result	A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers.

BSBlockListDescriptorGetExtent

Retrieves the address and length of the next extent in a descriptor, relative to the offset.

```
extern OSStatus BSBlockListDescriptorGetExtent (
                    BSBlockListDescriptorRef srcDescriptor,
                    ByteCount requestedLen,
                    BSByteCount *startingByte,
                    ByteCount *extentLen);
```

`srcDescriptor`	A reference to the block list descriptor (page 7-32) from which you want extent information.
`requestedLen`	The maximum number of bytes you can transfer to or from a device. To specify an unlimited number, set this parameter to 0. If the next extent exceeds your maximum, the function splits the extent into two or more extents and returns your maximum in the `extentLen` parameter.
`startingByte`	A pointer to a 64-bit value. On output, the function returns the starting address of the next extent.

`extentLen`	A pointer to a 32-bit value. On output, the function returns the length of the extent specified in the `startingByte` parameter, in bytes.
function result	A result code. If the function has already returned information on all the extents in a descriptor, it returns the `E_BSBLEndOfList` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSBlockListDescriptorGetExtent` function is used by mapping plug-ins that manage primary stores. When making an I/O request to a device, you typically want to transfer as many bytes as possible. Often, the device requires that

- the number of bytes transfered be equal to or less than a device-specific maximum
- all bytes transferred have contiguous addresses

A block list descriptor for an I/O transfer can contain any number of extents—variable-length sets of contiguous addresses. You call `BSBlockListDescriptorGetExtent` to get the address and length of an extent from the block list, subject to a maximum length that you specify.

You can then specify the returned extent information (starting device address and length) in the I/O request you send to the device, usually through another I/O family.

By calling `BSBlockListDescriptorGetExtent` repeatedly, you can get all of the extents specified in a block list descriptor. On the first call, the function returns the starting address and length of the first extent, and updates the offset, a private pointer into the descriptor that it maintains. On each subsequent call to `BSBlockListDescriptorGetExtent`, the function returns the starting address and length of the next extent, and updates the offset.

This method returns extents sequentially from the descriptor. To access an arbitrary point in the descriptor, you can call the `BSBlockListDescriptorSeek` function (page 7-155) and set the offset to a value you specify, and then call `BSBlockListDescriptorGetExtent`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

BSBlockListAddSimpleDescriptor

Creates a new descriptor for a block list.

```
extern OSStatus BSBlockListAddSimpleDescriptor (
                    BSBlockListDescriptorRef srcDescriptor,
                    BSByteCount length,
                    BSByteCount bias,
                    BSBlockListDescriptorRef *newDescriptor);
```

`srcDescriptor` A reference to the block list descriptor (page 7-32) on which the new descriptor will be based. You provide the reference.

`length` The number of bytes to be described by the new descriptor. The function creates a new descriptor for the number of bytes you specify here, starting from the offset. (The offset is a private pointer into the descriptor maintained by the block storage family.) To specify the same number of bytes as in the source descriptor, set this parameter to 0. In that case, the offset is irrelevant.

`bias` The value to add to the addresses in the source descriptor. You provide the value. The function adds the bias to the source addresses. A bias is always positive.

`newDescriptor` A pointer to a block list descriptor reference (page 7-32). On output, the function supplies a reference to the new block list descriptor.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

The `BSBlockListAddSimpleDescriptor` function is of central importance to a mapping plug-in that manages a derived store. The plug-in's main function is to map addresses associated with an I/O request from addresses valid in its store to addresses valid in the store to which the plug-in will forward the I/O request. It does this in its I/O function by calling `BSBlockListAddSimpleDescriptor`.

The `BSBlockListAddSimpleDescriptor` function creates a new descriptor and shifts the addresses by the amount you specify in the `bias` parameter. If you want to shift all of the addresses covered in the source descriptor, you set the `length` parameter to 0.

You can create a new descriptor that covers a portion of the addresses from the source descriptor by setting the `length` parameter to the number of bytes you want included in the new descriptor. The function shifts the addresses of the number of bytes you specified, starting at the current offset, and then updates the offset.

For example, suppose the source descriptor contains 2 ranges, with starting addresses of 1800 and 3000 and a total length of 1024. Then suppose you call `BSBlockListAddSimpleDescriptor` and set `bias` to 400 and `length` to 0. The function returns a reference to a new descriptor. The new descriptor contains 2 ranges, with starting addresses of 2200 and 3400 and a total length of 1024.

At times, you may need to split an I/O request—for example, perhaps the next store cannot accept I/O requests of more than 512 bytes. For simplicity's sake, assume that the 2 ranges in the example source descriptor are each 512 bytes in length. In this scenario, you can call `BSBlockListAddSimpleDescriptor` twice to create 2 descriptors, each describing a 512-byte transfer. Each time, you set `bias` to 400 and `length` to 512. The first new descriptor contains a single range, with a starting address of 2200 and a length of 512. The second new descriptor contains a single range, with a starting address of 3400 and a length of 512.

When you specify a length other than the total length of the source descriptor, the new descriptor begins at the current offset into the source descriptor. The block storage family maintains the offset. After a call to `BSBlockListAddSimpleDescriptor`, the offset points at the first byte after the end of those described by the new descriptor. In the example, after the first call to `BSBlockListAddSimpleDescriptor`, the offset was 512, which in effect was address 3400.

Although not shown in the example, splitting a source descriptor into can result in a range being split into two ranges—the last range on one descriptor and the first one on the next descriptor.

Note
A mapping plug-in can also map addresses by creating a new block list and descriptor.

Mapping plug-ins that manage primary stores create device-specific structures that contain the mapped addresses. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can change the offset into a descriptor with the `BSBlockListDescriptorSeek` function (page 7-155).

To create a new block list and descriptor, use the `BSBlockListCreate` (page 7-81), `BSBlockListAddRange` (page 7-82), and `BSBlockListFinalize` (page 7-84) functions.

BSBlockListDescriptorCheckBlockSizes

Given a block size, checks that the extents specified by a descriptor have valid starting addresses and lengths.

```
extern OSStatus BSBlockListDescriptorCheckBlockSizes (
                    BSBlockListDescriptorRef checkDescriptor,
                    UInt32 blockSize);
```

`checkDescriptor`
A reference to the block list descriptor (page 7-32) of interest.

`blockSize` The block size, in bytes.

function result A result code. If any extent is misaligned, the function returns the `E_BSBlockListBadBlockSize` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

Each store defines its own block size and all I/O requests to a store must use a multiple of the block size. A mapping plug-in's I/O function (page 7-183) can call the `BSBlockListDescriptorCheckBlockSizes` function to confirm that the extents specified by a block list descriptor are valid before it processes the I/O request.

The function checks that each extent's starting address and length are even multiples of the block size you provide in the `blockSize` parameter.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

BSBlockListDescriptorCheckBounds

Checks that the extents specified by a descriptor do not extend beyond an address that you specify.

```
extern OSStatus BSBlockListDescriptorCheckBounds (
                    BSBlockListDescriptorRef checkDescriptor,
                    BSByteCount bound);
```

`checkDescriptor`
A reference to the block list descriptor (page 7-32) of interest.

`bound` A 64-bit value that specified the highest valid address in a store.

function result A result code. If any extent is misaligned, the function returns the `E_BSBlockListBadBlockSize` result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

A mapping plug-in's I/O function (page 7-183) typically calls the `BSBlockListDescriptorCheckBounds` function before it processes an I/O request. The function confirms that no address covered by the block list descriptor is greater than the highest address in the store managed by the mapping plug-in.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

BSBlockListDescriptorSeek

Sets the offset within a block list descriptor.

```
extern OSStatus BSBlockListDescriptorSeek (
                    BSBlockListDescriptorRef seekDescriptor,
                    BSBlockListWhence whence,
                    SInt64 offset,
                    BSByteCount *newOffset);
```

`seekDescriptor`
A reference to the block list descriptor (page 7-32) of interest.

`whence` A value of type `BSBlockListWhence` (page 7-52) that indicates the method to be used in computing the new offset.

`offset` A number you provide that is used by the function to compute the new offset according to the method you specified in the `whence` parameter. See the description of the `BSBlockListWhence` type for guidance in setting this parameter.

`newOffset` A pointer to a 64-bit value. On output, the function returns the new offset, expressed as a byte address.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

When you call the `BSBlockListDescriptorGetExtent` function (page 7-148) to get an extent or the `BSBlockListAddSimpleDescriptor` function (page 7-150) to create a new block list descriptor, the results of the function depend on the current offset into the block list descriptor. The block storage family maintains the

offset into a block list descriptor. However, you can use the `BSBlockListDescriptorSeek` function to explicitly set the offset.

For example, in retrieving information about the extents in a descriptor, the usual case is to do it sequentially by calling `BSBlockListDescriptorGetExtent` repeatedly. The offset is automatically updated after each call so that the next call returns the next extent.

If you want to get an extent in a random-access manner, however, you can use the `BSBlockListDescriptorSeek` function to set the offset. Then, you can call `BSBlockListDescriptorGetExtent` to get a specific extext within a descriptor.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

BSBlockListDescriptorDelete

Disposes of a block list descriptor.

```
extern OSStatus BSBlockListDescriptorDelete (
                    BSBlockListDescriptorRef deleteDescriptor);
```

`deleteDescriptor`
A reference to the block list descriptor (page 7-32) that you want to delete.

function result A result code. See "Block Storage Result Codes" (page 7-205) for a list of the result codes the block storage family can return.

DISCUSSION

You call the `BSBlockListDescriptorDelete` function when you have finished processing the I/O request with which the block list descriptor is associated. The function releases all resources associated with a block list descriptor, including memory. If no block list descriptors remain for a block list, `BSBlockListDescriptorDelete` disposes of the block list also.

The client who makes an I/O request deletes the original descriptor. Mapping plug-ins that create a new descriptor based on an original descriptor delete those that they create.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers and secondary interrupt handlers.

SEE ALSO

The `BSBlockListDelete` function (page 7-85) also deletes a block list.

Block Storage Plug-in Functions

The functions in this section are those that the block storage family provides to its plug-ins and those that block storage plug-ins make available to the block storage family.

Some functions in the client interface are also called by plug-ins.

- Mapping plug-ins call the functions described in "Working With a Block List Descriptor" (page 7-147).
- Partitioning plug-ins call the I/O functions described in "Reading From a Store" (page 7-86) and "Writing To a Store" (page 7-92) to access partition maps.

Exported By the Block Storage Family For All Plug-ins

The block storage family provides the function described in this section for all its plug-ins, regardless of type.

BSStoreGetAccessibilityState

Returns the accessibility state of a store.

```
extern BSAccessibilityState BSStoreGetAccessibilityState (
                    BSStorePtr accessStore);
```

`accessStore` On input, a pointer to the store of interest.

function result The store's accessibility state. See "Accessibility State Type" (page 7-41) for descriptions of the defined states.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage plug-ins call this function. They can be any type—mapping, partition, or container plug-ins.

BSStoreGetMPIInfo

Gets information from a mapping plug-in about a store that you specify.

```
extern OSStatus BSStoreGetMPIInfo(
                    BSStorePtr accessStore,
                    BSStoreMPIInfo *info);
```

`accessStore` On input, a pointer to the store about which you want information.

`info` A pointer to a `BSStoreMPIInfo` structure (page 7-58) that you provide. On output, the function fills in the fields of the structure.

function result A result code.

DISCUSSION

When you call the `BSStoreGetMPIInfo` function, the block storage family calls the information function (page 7-190) of the mapping plug-in associated with the store you specify.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage plug-ins call this function. They can be any type—mapping, partition, or container plug-ins.

BSStoreGetPPIInfo

Gets information from a partitioning plug-in about a store that you specify.

```
extern OSStatus BSStoreGetPPIInfo(
                    BSStorePtr accessStore,
                    BSStorePPIInfo *info);
```

`accessStore` On input, a pointer to the store about which you want information.

`info` A pointer to a `BSStorePPIInfo` structure (page 7-59) that you provide. On output, the function fills in the fields of the structure.

function result A result code.

DISCUSSION

When you call the `BSStoreGetPPIInfo` function, the block storage family calls the information function (page 7-190) of the partitioning plug-in associated with the store you specify.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage plug-ins call this function. They can be any type—mapping, partition, or container plug-ins.

Exported by the Block Storage Family For Mapping Plug-ins

The block storage family provides the functions described in this section for its mapping plug-ins.

BSStoreRW

Makes an I/O request of another store.

```
extern BSIOStatus BSStoreRW (
                    BSStorePtr rwStore,
                    BSBlockListDescriptorRef blocks,
                    MemListDescriptorRef memory,
                    BSIORequestBlockPtr parentRequest,
                    void *privateData,
                    OptionBits options,
                    BSErrorListPtr *errors);
```

`rwStore` On input, a pointer to the store that is the target of the I/O request.

`blocks` A reference to a block list descriptor (page 7-32) that specifies the blocks you want to use in the I/O transfer.

`memory` A reference to a memory list descriptor that specifies the memory locations you want to use in the I/O transfer. For information on memory lists, see "Memory Lists", a chapter to be provided in a later Developer Release.

`parentRequest` A token that identifies the I/O request. You get the token from the block storage family when it calls your I/O function (`MyBSMappingPIIOFunc` (page 7-183)). Pass that token in this parameter.

`privateData` Reserved for your use. The value you provide is passed to your completion routine (`MyBSMappingIOCompletionFunc` (page 7-185)) when it is called.

`options` A constant you use to indicate whether the I/O request is a read or a write. The constants are defined in "I/O Constants" (page 7-50).

`errors` The address of a pointer to a `BSErrorList` structure (page 7-55). On output, the function uses the structure to return error information if an error occurs.

function result A value of type `BSIOStatus` (page 7-55). The function returns the I/O status code provided by the I/O function of the plug-in that manages the store specified in the `rwStore` parameter.

DISCUSSION

A mapping plug-in that manages a derived store calls the `BSStoreRW` function to forward an I/O request to another mapping plug-in.

Typically, it first calls the `BSBlockListDescriptorCheckBlockSizes` (page 7-153), `BSBlockListDescriptorCheckBounds` (page 7-154) , and `BSBlockListAddSimpleDescriptor` (page 7-150) functions to validate and adjust the store addresses used in the I/O request.

Mapping plug-ins that manage primary stores do not call `BSStoreRW`. Rather, they call into another I/O family to handle the I/O request.

The `BSStoreRW` function is asynchronous. The plug-in's completion routine is called when the I/O request completes.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins that manage derived stores call this function. It cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSTrackOtherFamilyRequest

Sets up the conditions that enable the block storage family to match an I/O completion notification from another I/O family with the mapping plug-in that made the I/O request.

```
extern OSStatus BSTrackOtherFamilyRequest (
                    BSStorePtr ioStore,
                    BSIORequestBlockPtr curRequest,
                    void *privateData,
                    KernelNotificationPtr retNotify);
```

`ioStore`	On input, a pointer to the store that is the target of the I/O request. You get the pointer from the block storage family when it calls your I/O function (`MyBSMappingPIIOFunc` (page 7-183)). Pass that pointer in this parameter.
`curRequest`	A token that identifies the I/O request to be tracked. You get the token from the block storage family when it calls your I/O function (`MyBSMappingPIIOFunc` (page 7-183)). Pass that token in this parameter.
`privateData`	Reserved for your use. The value you provide here is passed to your completion routine (`MyBSMappingIOCompletionFunc` (page 7-185)) when it is called.
`retNotify`	A pointer to a kernel notification structure. On output, the function supplies a filled-in structure that specifies how the block storage family wants to be notified when your I/O request is complete. You pass it on your asynchronous call to another I/O family.
function result	A result code.

DISCUSSION

Mapping plug-ins that manage primary stores call the `BSTrackOtherFamilyRequest` function before calling outside the block storage family to satisfy an I/O request.

Because asynchronous calls to other I/O families are beyond the knowledge and control of the block storage family, block storage uses `BSTrackOtherFamilyRequest` to set up an environment in which it can match an I/O completion notification from another I/O family with the mapping plug-in that initiated the I/O request.

When the block storage family is notified that a request is complete, it matches the request to the mapping plug-in that initiated it and calls the plug-in's completion routine.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins that manage primary stores call this function. It cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreFlush

Flushes a store's caches.

```
extern BSIOStatus BSStoreFlush(
                    BSStorePtr flushStore,
                    BSIORequestBlockPtr parentRequest,
                    void *privateData,
                    BSErrorListPtr *errors);
```

`flushStore` On input, a pointer to the store that is the target of the flush request.

`parentRequest`
A token that identifies the I/O request whose data is to be flushed. You get the token from the block storage family when it calls either your I/O function (`MyBSMappingPIIOFunc` (page 7-183)) or your flush function (`MyBSMappingPIFlushFunc` (page 7-186)). Pass that token in this parameter

`privateData` Reserved for your use. The value you provide is passed to your completion routine (`MyBSMappingIOCompletionFunc` (page 7-185)) when it is called.

`errors` The address of a pointer to a `BSErrorList` structure (page 7-55). On output, the function uses the structure to return error information if an error occurs.

function result The I/O status code provided by the flush function of the plug-in that manages the store specified in the `flushStore` parameter.

DISCUSSION

A mapping plug-in that manages a derived store calls the `BSStoreFlush` function to forward a flush request to another mapping plug-in.

When the flush request completes, the block storage family calls the plug-in's completion routine.

Mapping plug-ins that manage primary stores do not call `BSStoreFlush`. Rather, they call into another I/O family to handle the flush request.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins that manage derived stores call this function. This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSMPIStartBackgroundTask

Starts a mapping plug-in's background task.

```
extern OSStatus BSMPIStartBackgroundTask (
                    BSStorePtr store,
                    BSMPIBackgroundTask backgroundTask,
                    void *arg,
                    TaskID *taskID);
```

`store` On input, a pointer to the store managed by this plug-in.

`backgroundTask` The entrypoint of the background task you want to start. (page 7-191).

`arg` Reserved for your use. Whatever value you provide here is passed to the background task when it is started.

`taskID` A pointer to a task ID. On output, the function returns the task ID of the new task.

function result A result code.

DISCUSSION

The task started by calling the `BSMPIStartBackgroundTask` function is terminated after the block storage family calls the plug-in's clean up function (page 7-182) for the store specified in the `store` parameter.

A mapping plug-in can start any number of background tasks.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function. It cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSGetMappingPIPrivateData

Retrieves a mapping plug-in's private data.

```
extern void * BSGetMappingPIPrivateData (BSStorePtr accessStore);
```

`accessStore` On input, a pointer to the store managed by this plug-in.

function result A pointer to your private data.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function.

BSSetMappingPIPrivateData

Writes a mapping plug-in's private data to a store.

```
extern void BSSetMappingPIPrivateData (
                    BSStorePtr accessStore,
                    void *privateData);
```

`accessStore` On input, a pointer to the store managed by this plug-in.

`privateData` On input, a pointer to your private data.

function result None.

DISCUSSION

A mapping plug-in might use private data to keep track of the pieces of an I/O request. Because it consumes physically resident memory, you should keep private data to a minimum.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function.

BSMPINotifyFamilyStoreChangedState

Informs the block storage family of an unexpected change in the accessibility state of a store.

```
extern OSStatus BSMPINotifyFamilyStoreChangedState (
                    BSStorePtr changedStore,
                    BSAccessibilityState newState);
```

`changedStore` On input, a pointer to the store managed by this plug-in.

`newState` The new accessibility state of the store.

function result A result code.

DISCUSSION

A mapping plug-in typically calls this function from its background task. Unexpected accessibility state changes can occur when media is removed or a disk spins down.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function. It cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSMPIRequestStoreStateChange

Requests that the accessibility state of a store be allowed to change.

```
extern OSStatus BSMPIRequestStoreStateChange (
                    BSStorePtr changeStore,
                    BSAccessibilityState requestedState,
                    Boolean *permission);
```

`changeStore` On input, a pointer to the store managed by this plug-in.

`requestedState`
The accessibility state that you want the store to go to.

`permission` A pointer to a Boolean value. On output, the function sets the value to `true` if permission was granted and the store successfully went to the state requested. Otherwise, it sets it to `false`.

function result A result code.

DISCUSSION

If a device has an eject button that can be monitored by a mapping plug-in, then when the eject button is pressed, the plug-in should call the `BSMPIRequestStoreStateChange` function to notify the block storage family. The family can then notify its clients that an eject has been requested and see that data is flushed to the media properly.

The `BSMPIRequestStoreStateChange` function waits until permission is granted or denied by the block storage expert.

If permission is granted, the block storage family calls the plug-in's accessibility state function (`BSMappingPIGoToState` (page 7-188)) before `BSMPIRequestStoreStateChange` returns.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function. It cannot be called by hardware interrupt handlers or secondary interrupt handlers.

BSStoreGetNumComponents

Returns the number of component stores or devices that make up a store that you specify.

```
extern ItemCount BSStoreGetNumComponents (BSStorePtr accessStore);
```

`accessStore` A pointer to the store managed by this plug-in.

function result The number of components in the store.

DISCUSSION

A **component** is a constituent part of a store, the thing on which a given store is based. A component of a primary store is a device controlled by another I/O family. A component of a derived store is another store or a partition of a store.

Although most stores have only one component, some have multiple components. For example, a store representing a RAID system has multiple components.

Once you know the number of components in a store, you can call the `BSStoreGetComponent` function (page 7-171) as many times as necessary to get information about each component.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function.

BSStoreGetComponent

Gets information about a component of a store.

```
extern OSStatus BSStoreGetComponent (
                    BSStorePtr accessStore,
                    ItemCount componentNum,
                    BSStoreMPIComponentPtr component);
```

`accessStore` On input, a pointer to the store managed by this plug-in.

`componentNum` The sequence number of the component about which you want information. Component numbering starts at 0. You can get information about all components in a store by calling the `BSStoreGetNumComponents` function (page 7-170) to get the total number of components in a store and then calling this function for each component. Set the `componentNum` parameter to 0 on the first call and increment it by 1 on each subsequent call.

`component` A pointer to a `BSStoreMPIComponent` structure (page 7-57). On output, the function fills in information about the component.

function result A result code.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage mapping plug-ins call this function.

Exported by the Block Storage Family For Partitioning Plug-ins

The block storage family provides the functions described in this section for its partitioning plug-ins.

BSStoreSetNumPartitions

Sets the number of partitions in a store.

```
extern void BSStoreSetNumPartitions (
                    BSStorePtr accessStore,
                    ItemCount numPartitions);
```

`accessStore` On input, a pointer to the store whose number of partitions you want to set.

`numPartitions` The number of partitions to set.

function result None.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage partitioning plug-ins call this function.

BSGetPartitioningPIPrivateData

Retrieves a partitioning plug-in's private data from a store.

```
extern void * BSGetPartitioningPIPrivateData (BSStorePtr accessStore);
```

`accessStore` On input, a pointer to the store from which you want to retrieve your private data.

function result Your private data.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage partitioning plug-ins call this function.

BSSetPartitioningPIPrivateData

Sets a partitioning plug-in's private data for a store.

```
extern void BSSetPartitioningPIPrivateData (
                    BSStorePtr accessStore,
                    void *privateData);
```

`accessStore` On input, a pointer to the store for which you want to set your private data.

`privateData` On input, a pointer to your private data.

function result None.

DISCUSSION

•••To be provided•••

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage partitioning plug-ins call this function.

BSStoreGetPPIConnection

Returns a connection ID for a store.

```
extern BSStoreConnID BSStoreGetPPIConnection (BSStorePtr accessStore);
```

`accessStore` On input, a pointer to the store of interest.

function result A connection ID. If the function fails to get a connection ID, it returns the value `kInvalidID`.

DISCUSSION

When a client calls the `BSStoreConnAssociatePartitioningPlugin` function, an exclusive I/O connection to the store is opened for the partitioning plug-in so that it can read and write the partition map.

The plug-in retrieves the connection ID by calling `BSStoreGetPPIConnection`. Then it can call the functions described in "Reading From a Store" (page 7-86) and "Writing To a Store" (page 7-92) to read and write to the store.

(Note that a partitioning plug-in calls read and write functions in the client programming interface. It does not call the `BSStoreRW` function (page 7-161) for I/O as mapping plug-ins do.)

The connection stays open as long as the store exists.

Note
Block storage clients can read or write only to leaf stores. A store from which other stores are derived has a partitioning plug-in associated with it. The partitioning plug-in has an exclusive connection to the store.

Mapping plug-in, on the other hand, do not need connection IDs to access stores—they can write anywhere on any store. They are trusted to preserve the partition map and observe partition boundaries. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	Yes	Yes

CALLING RESTRICTIONS

Only block storage partitioning plug-ins call this function.

Exported by the Block Storage Family For Container Plug-ins

The block storage family provides the functions described in this section for its container plug-ins.

BSCPIStartBackgroundTask

Starts a standard background task for a container plug-in.

```
extern OSStatus BSCPIStartBackgroundTask(
                    BSContainerPtr container,
                    BSCPIBackgroundTask backgroundTask,
                    void *arg,
                    TaskID *taskID);
```

`container` On input, a pointer to the container for which the background task is being started.

`backgroundTask` The entrypoint of the standard background task you want to start (page 7-204).

`arg` Reserved for your use. Whatever value you provide here is passed to the background task when it is started.

`taskID` A pointer to a task ID. On output, the function returns the task ID of the new task.

function result A result code.

DISCUSSION

The task is automatically terminated after the block storage family calls the plug-in's cleanup function (page 7-200) for the container specified in the `container` parameter.

A container plug-in can start any number of standard background tasks.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage container plug-ins call this function.

BSCPINotifyFamilyContainerChangedState

Informs the block storage family of a change in the accessibility state of a container.

```
extern OSStatus BSCPINotifyFamilyContainerChangedState(
                    BSContainerPtr changedContainer,
                    BSAccessibilityState newState);
```

`changedContainer`
On input, a pointer to the container whose state has changed.

`newState` The current accessibility state of the container.

function result A result code.

DISCUSSION

A container plug-in typically calls this function from its background task. Accessibility state changes can occur when media is removed or a disk spins down.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage container plug-ins call this function.

BSCPIRequestContainerStateChange

Requests that the accessibility state of a container be allowed to change.

```
extern OSStatus BSCPIRequestContainerStateChange(
                    BSStorePtr changeContainer,
                    BSAccessibilityState requestedState,
                    Boolean *permission);
```

`changeContainer`
On input, a pointer to the container of interest.

`requestedState`
The accessibility state that you want the container to go to.

`permission` A pointer to a Boolean value. On output, the function sets the value to `true` if permission was granted and the container successfully went to the state requested. Otherwise, it sets it to `false`.

function result A result code.

DISCUSSION

If a device has an eject button that can be monitored by a container plug-in, then when the eject button is pressed, the plug-in should call the `BSCPIRequestStoreStateChange` function to notify the block storage family. The family can then notify its clients that an eject has been requested and see that data is flushed to the media properly.

The `BSCPIRequestStoreStateChange` function does not return until permission is granted or denied by the block storage expert.

If permission is granted, the block storage family calls the plug-in's accessibility state function (`BSContainerPIGoToState` (page 7-201)) before `BSCPIRequestStoreStateChange` returns.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

Only block storage container plug-ins call this function.

Mapping Plug-in-Defined Functions

A mapping plug-in provides the functions described in this section. The functions are called only by the block storage family or the block storage expert.

MyBSMappingPIExamineFunc

Queries a mapping plug-in about its ability to support a given store.

```
extern OSStatus MyBSMappingPIExamineFunc (
                    BSStorePtr examineStore,
                    BSMPIConfidenceLevel *confidence);
```

`examineStore` On input, a pointer to the store that the plug-in is to examine. This store is sometimes referered to as an *in-process store*—it is in the process of being configured by the block storage expert and is not yet published in the name registry and available for use.

`confidence` A pointer to a value of type `BSMPIConfidenceLevel`. On output, the plug-in returns a value indicating its level of confidence that it can support the store. See "BSMPIConfidenceLevel" (page 7-53) for a description of the confidence level values you can return here.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. Successful execution includes the plug-in determining that it cannot support the store. If an error occurs, it should return an appropriate result code. For example, if a mapping plug-in for a hard disk on a SCSI bus encounters an error while querying a device, it should return the result code it gets from the SCSI family.

DISCUSSION

When the block storage expert receives a notification of a new block storage entry in the name registry, it creates a new store and calls the examine function of each mapping plug-in to find out how well the plug-ins can support the store.

When this function is called, the mapping plug-in should examine the store specified in the `examineStore` parameter by

- calling the `BSStoreGetNumComponents` function (page 7-170) to find out how many components the store has
- calling the `BSStoreGetComponent` function (page 7-171) for each component to get information about the component
- examining each component. The plug-in can look at the information in the name registry, try an I/O operation with a physical component, or take whatever action it deems appropriate to assess its ability to support the component.

Then the plug-in decides on its level of confidence that it can support the store and returns that information in the `confidence` parameter.

Before returning, your examine function must free any resources it allocates. The block storage expert selects the plug-in that offers the highest level of support. If the expert selects a different plug-in to manage the store, your plug-in does not get another chance to free resources.

Note
The following example may be helpful in understanding the circumstances under which the examine function is called.

Suppose the SCSI expert probes a SCSI bus and finds a disk drive. It adds an entry for the device to the name registry and sends a new device notification. The system matches the device to block storage plug-ins and sends a new block storage device notification. The block storage expert responds by preparing a new store and calling the examine routine of each mapping plug-in. The plug-in reports how well it can support the device. The expert selects a plug-in to manage the store, configures the store, publishes it in the name registry, and calls the initialization function of the mapping plug-in it selected. Once the plug-in's initialization function returns with no error, the store is available to block storage clients for I/O. ◆

The `BSMappingPIExamine` type (page 7-65) defines the mapping plug-in examine function.

SPECIAL CONSIDERATIONS

Only the block storage expert calls this function.

MyBSMappingPIInitFunc

Initializes a mapping plug-in.

```
extern OSStatus MyBSMappingPIInitFunc (
                    BSStorePtr initStore);
```

`initStore` On input, a pointer to the store that the mapping plug-in is to manage.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Mapping Plug-in Errors" (page 7-207) for a list of the result codes a mapping plug-in can return.

DISCUSSION

The block storage family calls a mapping plug-in's initialization function after it selects the plug-in to manage a store. Your plug-in should perform whatever setup work is necessary, such as initializing a device, connecting to another I/O family, or allocating memory.

The `BSMappingPIInit` type (page 7-66) defines the mapping plug-in initialization function.

SEE ALSO

Any resources that you acquire during initialization should be released in your clean up function(page 7-182).

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPICleanupFunc

Completes processing and releases resources for a store.

```
extern OSStatus MyBSMappingPICleanupFunc (
                    BSStorePtr cleanupStore);
```

`cleanupStore` On input, a pointer to the store managed by this plug-in.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Mapping Plug-in Errors" (page 7-207) for a list of the result codes a mapping plug-in can return.

DISCUSSION

The block storage family calls a mapping plug-in's clean up function prior to disposing of a store. Your plug-in should perform whatever work is necessary, such as disposing of memory and data structures related to the store and completing all I/O requests still outstanding.

Be sure to release any resources that you acquired in your initialization function (page 7-181).

The `BSMappingPICleanup` type (page 7-66) defines the mapping plug-in clean up function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIIOFunc

Processes an I/O request.

```
extern BSIOStatus MyBSMappingPIIO (
                    BSStorePtr ioStore,
                    BSBlockListDescriptorRef blocks,
                    MemListDescriptorRef memory,
                    BSIORequestBlockPtr parentRequest,
                    OptionBits options,
                    BSErrorList **errors);
```

`ioStore` On input, a pointer to the store managed by this plug-in.

`blocks` A reference to a block list descriptor (page 7-32) that specifies the address ranges in the store to be read from or written to.

`memory` A reference to a memory list descriptor that specifies the byte addresses in memory to be read from or written to. For information on memory lists, see “Memory Lists”, a chapter to be provided in a later Developer Release.

`parentRequest`	A token provided by the block storage family that identifies this I/O request. You pass the token to the `BSStoreRW` (page 7-161) or `BSTrackOtherFamilyRequest` (page 7-162) function as you process the I/O request.
`options`	A constant that specifies whether the I/O request is a read or a write request. The constants are defined in "I/O Constants" (page 7-50).
`errors`	The address of a pointer to an `BSErrorList` structure (page 7-55).
function result	An I/O status code. "BSIOStatus" (page 7-55) describes defined status codes.

DISCUSSION

The I/O function of a mapping plug-in is responsible for satisfying an I/O request presented to it by the block storage family.

If the plug-in manages a derived store, it needs to map the address ranges associated with the request into corresponding ranges on a parent store and then call the `BSStoreRW` function (page 7-161) to forward the request. Address ranges for the I/O are specified in the block list descriptor referenced by the `blocks` parameter.

If the plug-in manages a primary store, it first needs to call the `BSTrackOtherFamilyRequest` function (page 7-162). That allows the block storage family to associate a future I/O completion notification with the plug-in.

Then, the plug-in needs to convert the request into a form valid to the next target software entity and then call that entity. For example, if a mapping plug-in for a primary store is a SCSI disk driver, it sets up a SCSI family function call and calls the SCSI family with its I/O request.

At the time your I/O function is called, the memory specified by the memory list descriptor is already prepared for I/O.

The `BSMappingPIIO` type (page 7-66) defines the mapping plug-in I/O function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingIOCompletionFunc

Activates a mapping plug-in's I/O completion routine.

```
extern OSStatus MyBSMappingIOCompletionFunc (
                    BSStorePtr theStore,
                    void *finishedPrivateData,
                    BSErrorListPtr returnedBSErrorList,
                    OSStatus returnedStatus,
                    BSErrorListPtr *errorListPtrPtr);
```

`theStore` On input, a pointer to the store managed by this plug-in.

`finishedPrivateData`
On input, a pointer to the mapping plug-in's private data for this I/O request.

`returnedBSErrorList`
On input, if the I/O request was signaled by another block storage plug-in and an error occurred, a pointer to a `BSErrorList` structure (page 7-56). Otherwise, this parameter contains `nil`.

`returnedStatus`
The `OSStatus` value returned by the I/O family or the block storage plug-in that serviced the I/O request. If this parameter contains a block storage family result code other than `E_BSSuccess`, the `returnedBSErrorList` parameter contains a pointer to more specific error information.

`errorListPtrPtr`
A pointer to a pointer to a `BSErrorList` structure. If the I/O request failed, the plug-in allocates the error list structure and, on output, fills it in. The block storage family releases the memory taken for the error list structure.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If a system-level error occurs, such as running out of memory or a failure to send a message, it should return an appropriate result code.

DISCUSSION

The block storage family calls your completion routine when it is notified that an I/O request is complete. You start an I/O request to another store by calling the `BSStoreRW` function (page 7-161). If you are a primary store plug-in, the method for starting an I/O request is defined by another I/O family or the device itself. In any case, the block storage family is notified when the request completes and calls your completion routine.

The `BSMappingIOCompletion` type (page 7-67) defines a mapping plug-in's I/O completion routine.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIFlushFunc

Flushes a mapping plug-in's cached data.

```
extern BSIOStatus MyBSMappingPIFlushFunc(
                    BSStorePtr ioStore,
                    BSIORequestBlockPtr parentRequest,
                    BSErrorList **errors);
```

`ioStore` On input, a pointer to the store of interest.

`parentRequest` A token provided by the block storage family that identifies the I/O request whose data is to be flushed. If, as you process the flush request, you call the `BSStoreRW` (page 7-161) or the `BSFlush` (page 7-164) function, you pass those functions the token provided here,

`errors` A pointer to a pointer to a `BSErrorList` structure. If the flush request fails, the plug-in allocates the error list structure and, on output, fills it in. The block storage family releases the memory used by the error list structure.

function result A result code.

DISCUSSION

The block storage family calls a mapping plug-in's flush function when a client or another mapping plug-in has requested that the store be flushed.

The `BSMappingPIFlush` type (page 7-67) defines the mapping plug-in flush function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIAddComponentFunc

Adds a component to a store.

```
extern OSStatus MyBSMappingPIAddComponentFunc (
                    BSStorePtr destStore,
                    BSStoreMPIComponent *newComponent,
                    BSStoreInfo *storeNewInfo);
```

`destStore` On input, a pointer to the store managed by this plug-in.

`newComponent` On input, a pointer to a store component structure (page 7-57) containing information about the component to be added.

`storeNewInfo` A pointer to a `BSStoreInfo` structure (page 7-44). On output, the plug-in supplies updated information about the store in the structure.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If it cannot add a component or if it has already added the maximum number of components that it can support, it should return the `E_BSMPITooManyMappings` result code. If the starting address of the new component would cause a gap in the addresses within a store, it should return the `E_BSMPIOutOfStoreBounds` result code.

DISCUSSION

This function allows a mapping plug-in to add a component to an existing store. For example, a RAID mapping plug-in might be able to integrate a new device into the RAID system. Or a plug-in might have the ability to concatenate a new device to an existing store, such as when a user connects a new drive to a system.

Not all plug-ins support this ability.

The `BSMappingPIAddComponent` type (page 7-68) defines the mapping plug-in add component function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIGoToStateFunc

Takes a store to a specified accessibility state.

```
extern OSStatus MyBSMappingPIGoToStateFunc (
                    BSStorePtr theStore,
                    BSAccessibilityState gotoState);
```

`theStore`	On input, a pointer to the store managed by this plug-in.
`gotoState`	The new accessibility state to be applied to the store.
function result	Your plug-in should return the result code `E_BSSuccess` to indicate successful execution.

DISCUSSION

Typically, the block storage family calls this function after the mapping plug-in has called the `BSMPIRequestStoreStateChange` function (page 7-169) and permission to change the accessibility state is granted by the block storage family expert.

The `BSMappingPIGoToState` type (page 7-68) defines the mapping plug-in accessibility state function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIFormatMediaFunc

Formats the media of a store.

```
extern OSStatus MyBSMappingPIFormatMediaFunc (
                    BSStorePtr formatStore,
                    BSFormatIndex formatType);
```

`formatStore` On input, a pointer to the store managed by this plug-in.

`formatType` A value that identifies the format to be applied.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Mapping Plug-in Errors" (page 7-207) for a list of the result codes a mapping plug-in can return.

DISCUSSION

When the format media function is called, the plug-in should do whatever is necessary to accomplish the formatting—a SCSI disk driver might issue the SCSI format command, a floppy disk driver might initialize the disk and set track and sector information on it, and so forth.

Although the preceding examples featured plug-ins for primary stores, plug-ins managing derived stores might also need to do something in their format media functions. For example, a RAID driver might need to set parity information on each of the components in the store it manages. In most cases, however, plug-ins managing derived stores would simply return with the `E_BSSuccess` result code.

The block storage family calls this function when a client calls the `BSStoreConnFormat` function (page 7-125).

The `BSMappingPIFormatMedia` type (page 7-69) defines the mapping plug-in format media function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMappingPIGetInfoFunc

Returns information about a store and its associated mapping plug-in.

```
extern OSStatus MyBSMappingPIGetInfoFunc (
                    BSStorePtr infoStore,
                    BSStoreMPIInfo *info);
```

`infoStore` On input, a pointer to the store managed by the plug-in.

`info` A pointer to a `BSStoreMPIInfo` structure (page 7-58). On output, the plug-in provides information about the store in the structure.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Mapping Plug-in Errors" (page 7-207) for a list of the result codes a mapping plug-in can return.

DISCUSSION

The plug-in can call the `BSStoreGetMPIInfo` function (page 7-159) to get information on a parent store if necessary.

The `BSMappingPIGetInfo` type (page 7-69) defines the mapping plug-in information function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSMPIBackgroundTaskFunc

Activates a mapping plug-in's background task.

```
extern OSStatus MyBSMPIBackgroundTaskFunc (
                    BSStorePtr theStore,
                    void *theArg);
```

`theStore` On input, a pointer to the store managed by the plug-in.

`theArg` Reserved for your use. The block storage family sets this parameter to the value that you provided when you called the `BSMPIStartBackgroundTask` function.

function result A result code. The plug-in is not notified if the function terminates with an error. However, the block storage family logs the result code with the system logging service.

DISCUSSION

When a mapping plug-in calls the `BSMPIStartBackgroundTask` function (page 7-165), the block storage family in turn calls the specified background task function.

The task is automatically terminated after the plug-in's clean up function is executed.

Because the plug-in is not notified if the background task terminates with an error, the background task should not terminate with an error, but rather do its own error handling.

Typically, you use a background task to monitor device states and to notify the block storage family of unexpected changes in accessibility state. Some plug-ins may have other uses for background tasks. For example, a RAID plug-in might use a background task to rebuild parity on a disk that was swapped into a disk array.

A plug-in can start any number of background tasks.

The `BSMPIBackgroundTask` type (page 7-70) defines a mapping plug-in's background task function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

Partitioning Plug-in-Defined Functions

A partitioning plug-in provides the functions described in this section. The functions are called only by the block storage family or the block storage expert.

MyBSPartitioningPIExamineFunc

Queries a partitioning plug-in about its ability to support a given store.

```
extern OSStatus MyBSPartitioningPIExamineFunc (
                    BSStoreConnID readStoreConn,
                    UInt32 *certainty);
```

`readStoreConn` The connection ID for the store to be examined.

`certainty` A pointer to a 32-bit value. On output, the plug-in returns 0 if it does not recognize the partition map format. Otherwise, it returns the number of bytes of the partition map that it read to make its determination. The block storage expert interprets higher values to indicate greater ability to support a store, relative to lower values.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Partitioning Plug-in Errors" (page 7-208) for a list of the result codes a partitioning plug-in can return.

DISCUSSION

The `BSPartitioningPIExamine` type (page 7-70) defines a partitioning plug-in's examine function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPIInitFunc

Initializes a partitioning plug-in.

```
extern OSStatus MyBSPartitioningPIInit (
                    BSStorePtr initStore);
```

`initStore` On input, a pointer to the store that the partitioning plug-in is to is to be associated with.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Partitioning Plug-in Errors" (page 7-208) for a list of the result codes a partitioning plug-in can return.

DISCUSSION

The block storage family calls a partitioning plug-in's initialization function when it selects the plug-in for a store. Your plug-in should perform whatever work is necessary, such as allocating memory.

The `BSPartitioningPIInit` type (page 7-71) defines the partitioning plug-in initialization function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPICleanupFunc

Releases resources and flushes the partition map for a store.

```
extern void MyBSPartitioningPICleanupFunc (BSStorePtr cleanupStore);
```

`cleanupStore` A pointer to the store being disposed of.

function result None.

DISCUSSION

The block storage family calls a partitioning plug-in's clean up function prior to disposing of a store. Your plug-in should perform whatever work it deems necessary, such as disposing of memory and data structures related to the store and flushing the partition map.

The `BSPartitioningPICleanup` type (page 7-71) defines the partitioning plug-in clean up function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPIInitializeMapFunc

Creates a new partition map for a store.

```
extern OSStatus BSPartitioningPIInitializeMap (BSStorePtr initStore);
```

`initStore` On input, a pointer to the store of interest.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Partitioning Plug-in Errors" (page 7-208) for a list of the result codes a partitioning plug-in can return.

DISCUSSION

The block storage family calls a partitioning plug-in's initialize map function after calling the plug-in's initialization function. It gives the plug-in an opportunity to overwrite an existing partition map and to use a new partitioning format for the store.

The `BSPartitioningPIInitializeMap` type (page 7-72) defines the partitioning plug-in initialize map function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPIGetInfoFunc

Retrieves information about a partition map and the associated partitioning plug-in.

```
extern OSStatus MyBSPartitioningPIGetInfoFunc (
                    BSStorePtr store,
                    BSStorePPIInfo *info);
```

`store`	On input, a pointer to the store of interest.
`info`	A pointer to a `BSStorePPIInfo` structure (page 7-59). On output, the plug-in provides in the structure information about the store's partition map.
function result	Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Partitioning Plug-in Errors" (page 7-208) for a list of the result codes a partitioning plug-in can return.

DISCUSSION

The `BSPartitioningPIGetInfo` type (page 7-72) defines the partitioning plug-in information function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPIGetEntryFunc

Retrieves a specified partition map entry.

```
extern OSStatus MyBSPartitioningPIGetEntryFunc (
                    BSStorePtr readStore,
                    ItemCount entryNum,
                    BSPartitionDescriptor *retEntry);
```

`readStore` On input, a pointer to the store of interest.

`entryNum` The ordinal number of the partition map entry to be read. Partition map entry numbering starts at 0.

`retEntry` A pointer to a `BSPartitionDescriptor` structure (page 7-48). On output, the plug-in fills in the structure.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. For example, if the specified entry does not exist. return the `E_BSPPIPartitionNonExistant` result code.

DISCUSSION

The `BSPartitioningPIGetEntry` type (page 7-72) defines the partitioning plug-in get entry function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSPartitioningPISetEntryFunc

Creates or modifies a partition map entry and the corresponding partition.

```
extern OSStatus MyBSPartitioningPISetEntryFunc (
                    BSStorePtr store,
                    ItemCount partitionNum,
                    BSPartitionDescriptor *partitionInfo);
```

`store` On input, a pointer to the store of interest.

`partitionNum` The ordinal number of the partition to be created or modified. Partition numbering starts at 0. A partition is defined by a partition map entry with the same ordinal number.

`partitionInfo` On input, a pointer to a `BSPartitionDescriptor` structure (page 7-48) that describes the partition.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Partitioning Plug-in Errors" (page 7-208) for a list of the result codes a partitioning plug-in can return.

DISCUSSION

A partitioning plug-in defines or modifies a partition by writing information in a partition map entry.

When a disk setup application calls the `BSStoreConnSetPartitionInfo` function (page 7-118), the block storage family responds by calling the set entry function of the appropriate partitioning plug-in to define the partition.

The `BSPartitioningPISetEntry` type (page 7-73) defines the partitioning plug-in set entry function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

Container Plug-in-Defined Functions

A container plug-in provides the functions described in this section. The functions are called only by the block storage family or the block storage expert.

MyBSContainerPIExamineFunc

Queries a container plug-in about its ability to support a given container.

```
extern OSStatus MyBSContainerPIExamineFunc (
                    BSContainerPtr initContainer,
                    BSCPIConfidenceLevel *confidence);
```

`initContainer` On input, a pointer to the container that the plug-in is to examine. This container is sometimes referered to as an *in-process container*—it is in the process of being configured by the block storage expert and is not yet published in the name registry and available for use.

`confidence` A pointer to a value of type `BSCPIConfidenceLevel`. On output, the plug-in returns a value indicating its level of confidence that it can support the device. See "BSCPIConfidenceLevel" (page 7-54) for a description of the confidence level values you can return here.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Container Plug-in Errors" (page 7-208) for a list of the result codes a container plug-in can return.

DISCUSSION

When the block storage expert receives a notification of a new block storage container entry in the name registry, it creates a new container and calls the examine function of each container plug-in to find out how well the plug-ins can support the container.

When this function is called, the container plug-in should examine the container specified in the `initContainer` parameter.

Then the plug-in decides on its level of confidence that it can support the container and returns that information in the `confidence` parameter.

Before returning, your examine function must free any resources it allocates. The block storage expert selects the plug-in that offers the highest level of support. If the expert selects a different plug-in to manage the container, your plug-in does not get another chance to free resources.

The `BSContainerPIExamine` type (page 7-74) defines the container plug-in examine function.

SPECIAL CONSIDERATIONS

Only the block storage expert calls this function.

MyBSContainerPIInitFunc

Initializes a container plug-in.

```
extern OSStatus MyBSContainerPIInitFunc (
                    BSContainerPtr initContainer
                    BSContainerPIInfo *info,
                    Boolean *backgroundTask);
```

`initContainer` On input, a pointer to the container that the container plug-in is to manage.

`info` A pointer to a container information structure (page 7-60). On output, the plug-in uses the structure to supply information about the container.

`backgroundTask`
A pointer to a Boolean value. On output, the plug-in sets the valueto `true` if it wants the block storage family to call its initialization background task function (page 7-76). Otherwise, it sets the value to `false`.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Container Plug-in Errors" (page 7-208) for a list of the result codes a container plug-in can return.

DISCUSSION

The block storage family calls a container plug-in's initialization function after it selects the plug-in to manage a container. Your plug-in should perform whatever setup work is necessary, such as allocating memory.

The `BSContainerPIInit` type (page 7-74) defines the container plug-in initialization function.

SEE ALSO

Any resources that you acquire during initialization should be released in your clean up function (page 7-75).

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSContainerPICleanupFunc

Completes processing and releases resources for a container.

```
extern OSStatus MyBSContainerPICleanupFunc (
                    BSContainerPtr container);
```

`container` On input, a pointer to the container managed by this plug-in.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Container Plug-in Errors" (page 7-208) for a list of the result codes a container plug-in can return.

DISCUSSION

The block storage family calls a container plug-in's clean up function prior to disposing of a container. Your plug-in should perform whatever work is necessary, such as disposing of memory and data structures related to the container.

Be sure to release any resources that you acquired in your initialization function (page 7-199).

The `BSContainerPICleanup` type (page 7-75) defines a container plug-in clean up function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSContainerPIGoToStateFunc

Takes a container to a specified accessibility state.

```
extern OSStatus MyBSContainerPIGoToStateFunc (
                    BSContainerPtr container,
                    UInt32 accessState);
```

`container`	On input, a pointer to the container managed by this plug-in.
`accessState`	The new accessibility state to be applied to the container.
function result	Your plug-in should return the result code `E_BSSuccess` to indicate successful execution.

DISCUSSION

Typically, the block storage family calls this function after the container plug-in has called the `BSCPIRequestContainerStateChange` function (page 7-178) and permission to change the accessibility state is granted by the block storage family expert.

The `BSContainerPIGoToState` type (page 7-75) defines a container plug-in accessibility state function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSContainerPIGetInfoFunc

Returns information about a container and its associated container plug-in.

```
extern OSStatus MyBSContainerPIGetInfoFunc (
                    BSContainerPtr infoContainer,
                    BSContainerPIInfo *info);
```

`infoContainer` On input, a pointer to the container managed by the plug-in.

`info` A pointer to a `BSContainerPIInfo` structure (page 7-60). On output, the plug-in provides information about the container in the structure.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution. If an error occurs, it should return an appropriate result code. See "Container Plug-in Errors" (page 7-208) for a list of the result codes a container plug-in can return.

DISCUSSION

The `BSContainerPIGetInfo` type (page 7-76) defines a container plug-in information function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSContainerPIAddContainerFunc

Adds a container to another container.

```
extern OSStatus MyBSContainerPIAddContainerFunc (
                    BSContainerPtr destContainer,
                    BSContainerPtr addedContainer);
```

`destContainer`
On input, a pointer to the container to which a container is being added.

`addedContainer`
On input, a pointer to the container to be added.

function result Your plug-in should return the result code `E_BSSuccess` to indicate successful execution

DISCUSSION

This function allows a container plug-in to add a container to an existing container.

Not all plug-ins support this ability.

The `BSContainerPIAddContainer` type (page 7-75) defines the container plug-in add container function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSContainerPIBackgroundTaskFunc

Activates a container plug-in's initialization background task.

```
extern OSStatus MyBSContainerPIBackgroundTaskFunc (
                    BSContainerPtr theContainer);
```

`theContainer` On input, a pointer to the container managed by the plug-in.

function result A result code. The plug-in is not notified if the function terminates with an error. However, the block storage family logs the result code with the system logging service.

DISCUSSION

If a container plug-in returns from its initialization function (page 7-199) with the `backgroundTask` flag set, the block storage family calls its initialization background task function.

The task is automatically terminated after the plug-in's clean up function is executed.

Because the plug-in is not notified if the task terminates with an error, the task should not terminate with an error, but rather do its own error handling.

The `BSContainerPIBackgroundTask` type (page 7-76) defines a container plug-in's initialization background task function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

MyBSCPIBackgroundTaskFunc

Activates a container plug-in's standard background task.

```
extern OSStatus MyBSCPIBackgroundTaskFunc  (
                    BSContainerPtr theContainer,
                    void *theArg);
```

`theContainer` On input, a pointer to the container for which the background task is being started.

`theArg` Reserved for your use. The block storage family sets this parameter to the value that you provided when you called the `BSCPIStartBackgroundTask` function (page 7-176).

function result A result code. The plug-in is not notified if the function terminates with an error. However, the block storage family logs the result code with the system logging service.

DISCUSSION

If a container plug-in calls the `BSCPIStartBackgroundTask` function, the block storage family responds by calling the standard background task function the plug-in specified. The task is automatically terminated after the plug-in's clean up function is executed.

Because the plug-in is not notified if the standard background task terminates with an error, the task should not terminate with an error, but rather do its own error handling.

A plug-in can start any number of standard background tasks.

The `BSCPIBackgroundTask` type (page 7-77) defines a container plug-in's standard background task function.

SPECIAL CONSIDERATIONS

Only the block storage family calls this function.

Block Storage Result Codes

•••To be provided•••

Basic Error Types

•••To be provided•••

```
enum {
    E_Success           = 0x00000000,
    E_LoopTermination   = 0x08000000,
    E_Underflow         = 0x10000000,
    E_Overflow          = 0x18000000,
    E_AlreadyExists     = 0x20000000,
    E_NotFound          = 0x28000000,
    E_AccessViolation   = 0x30000000,
    E_Busy              = 0x38000000,
    E_VersionMismatch   = 0x40000000,
    E_Canceled          = 0x48000000,
```

```
    E_OutOfResources    = 0x50000000,
    E_Timeout           = 0x58000000,
    E_ParameterError    = 0x60000000,
    E_Fatal             = 0x68000000,
    E_Unknown           = 0xF8000000
};
```

Block Storage Error ID

•••To be provided•••

```
enum {
    E_BlockStorageBias  = 0x04F00000
};
```

Block Storage Error Categories

•••To be provided•••

```
enum {
    E_BSFamilyError                 = 0x00000000 | E_BlockStorageBias,
    E_BSExpertError                 = 0x00008000 | E_BlockStorageBias,
    E_BSMappingPlugInError          = 0x00010000 | E_BlockStorageBias,
    E_BSPartitioningPlugInError     = 0x00018000 | E_BlockStorageBias,
    E_BSContainerPlugInError        = 0x00020000 | E_BlockStorageBias,
    E_BSBlockListError              = 0x00028000 | E_BlockStorageBias,
    E_BSSuccess                     = E_Success
};
```

Block Storage Family Errors

•••To be provided•••

```
enum {
    E_BSOutOfResources          = 0x00000001 | E_OutOfResources | E_BSFamilyError,
    E_BSStoreInUse              = 0x00000001 | E_AccessViolation | E_BSFamilyError,
    E_BSStoreWriteProtected     = 0x00000002 | E_AccessViolation | E_BSFamilyError,
    E_BSStoreNotFound           = 0x00000001 | E_NotFound | E_BSFamilyError,
```

```
    E_BSBadMessage              = 0x00000001 | E_ParameterError | E_BSFamilyError,
    E_BSBadConnection           = 0x00000002 | E_ParameterError | E_BSFamilyError,
    E_BSTableTooSmall           = 0x00000003 | E_ParameterError | E_BSFamilyError,
    E_BSNullParameters          = 0x00000004 | E_ParameterError | E_BSFamilyError,
    E_BSParameterError          = 0x00000005 | E_ParameterError | E_BSFamilyError,
    E_BSUnimplemented           = 0x00000001 | E_VersionMismatch | E_BSFamilyError
};
```

Block Storage Expert Errors

•••To be provided•••

```
enum {
    E_BSEPlugInNotFound     = 0x00000001 | E_NotFound | E_BSExpertError,
    E_BSENoPlugInMatch      = 0x00000002 | E_NotFound | E_BSExpertError,
    E_BSENoMoreStores       = 0x00000001 | E_OutOfResources | E_BSExpertError,
    E_BSEHierarchyTooDeep   = 0x00000002 | E_OutOfResources | E_BSExpertError,
    E_BSEOutOfResources     = 0x00000003 | E_OutOfResources | E_BSExpertError
};
```

Mapping Plug-in Errors

•••To be provided•••

```
enum {
    E_BSMPIOutOfStoreBounds = 0x00000001 | E_ParameterError | E_BSMappingPlugInError,
    E_BSMPITooManyMappings  = 0x00000002 | E_ParameterError | E_BSMappingPlugInError,
    E_BSMPIBadMappingParams = 0x00000003 | E_ParameterError | E_BSMappingPlugInError,
    E_BSMPIMappingNotSupported
                            = 0x00000004 | E_ParameterError | E_BSMappingPlugInError,
    E_BSMPIStateNotSupported
                            = 0x00000005 | E_ParameterError | E_BSMappingPlugInError,
    E_BSMPIWriteProtected   = 0x00000001 | E_AccessViolation | E_BSMappingPlugInError,
    E_BSMPICannotGoToState  = 0x00000002 | E_AccessViolation | E_BSMappingPlugInError,
    E_BSMPIUnitNotResponding
                            = 0x00000001 | E_Fatal | E_BSMappingPlugInError,
    E_BSMPITransferError    = 0x00000002 | E_Fatal | E_BSMappingPlugInError,
    E_BSMPIMemoryAccessFault
                            = 0x00000003 | E_Fatal | E_BSMappingPlugInError,
```

```
    E_BSMPINoPlugIn         = 0x00000004 | E_Fatal | E_BSMappingPlugInError,
    E_BSMPIMediaRemoved     = 0x00000005 | E_Fatal | E_BSMappingPlugInError,
    E_BSMPIOutOfResources   = 0x00000001 | E_OutOfResources | E_BSMappingPlugInError
};
```

Partitioning Plug-in Errors

•••To be provided•••

```
enum {
    E_BSPPIMappingNotSupported
                        = 0x00000001 | E_ParameterError | E_BSPartitioningPlugInError,
    E_BSPPIOverlappingPartition
                        = 0x00000002 | E_ParameterError | E_BSPartitioningPlugInError,
    E_BSPPIOutOfStoreBounds
                        = 0x00000003 | E_ParameterError | E_BSPartitioningPlugInError,
    E_BSPPIPartitionNonExistant
                        = 0x00000004 | E_ParameterError | E_BSPartitioningPlugInError,
    E_BSPPITooManyPartitions
                        = 0x00000005 | E_ParameterError | E_BSPartitioningPlugInError,
    E_BSPPINoPlugIn     = 0x00000001 | E_Fatal | E_BSPartitioningPlugInError,
    E_BSPPIOutOfResources
                        = 0x00000002 | E_OutOfResources | E_BSPartitioningPlugInError
};
```

Container Plug-in Errors

•••To be provided•••

Block List Errors

•••To be provided•••

```
enum {
    E_BSBLEndOfList         = 0x00000001 | E_Underflow | E_BSBlockListError,
    E_BSBLParameterError    = 0x00000001 | E_ParameterError | E_BSBlockListError,
    E_BSBLBadBlockList      = 0x00000002 | E_ParameterError | E_BSBlockListError,
    E_BSBLBadBlock          = 0x00000003 | E_ParameterError | E_BSBlockListError,
```

```
    E_BSBLAlreadyFinalized  = 0x00000001 | E_AlreadyExists | E_BSBlockListError,
    E_BSBLOutOfResources    = 0x00000001 | E_OutOfResources | E_BSBlockListError,
};
```

Glossary

accessibility state A condition that indicates how readily a block storage container or store can be accessed. Accessibility states range from fully accessible (online) to completely inaccessible (offline). More time is needed to access a container or store in a less accessible state, such as a disk drive in power-saving mode, than to access one in a more accessible state.

bias The positive value added to an address on a store to translate it to the equivalent address on another store.

block The minimum unit of I/O for a store. A fixed-length contiguous set of bytes within a store. See also **block size.**

block list An opaque data structure that specifies the address ranges on a store to be used for a given I/O transfer.

block list descriptor An opaque data structure that specifies a view of a block list. A view of a block list consists of a bias and a set of addresses. A single block list can have many descriptors with different biases and address ranges.

block size The number of bytes in a block. A store defines its read block size and write block size, which typically are the same, but may be different.

block storage family That part of the I/O system that abstracts the characteristics of, and operations on, large-capacity random access physical storage devices to provide a single and consistent interface to virtual storage devices. The block storage family keeps track of media and devices that can be controlled directly by Mac OS 8. The interfaces it provides include those to partition and aggregate physical storage devices into virtual devices, to mount and dismount volumes, and to control automated volume changers and ejectable media.

block storage plug-in See **container plug-in, mapping plug-in, partitioning plug-in.**

child store Relative to a given I/O request flowing between two stores, the store whose mapping plug-in translates addresses in order to read from or

write data to a parent store, and then forwards the request to the parent store. Compare **parent store.**

component A constituent part of a store, the thing on which a given store is based. A component of a primary store is a device controlled by another I/O family. A component of a derived store is another store or a partition of a store. A store may have more than one component, as a store representing a RAID system does.

connection A logical path to a store or a container. A connection serves to control access to a store or container and it is allocated as a result of opening a container or store. All operations that modify a store or a container require a connection. See also **control connection**, **I/O connection**.

connection ID A value that uniquely identifies a connection. It is assigned by Mac OS 8 when a new connection is created.

container An abstraction for hardware under the control of the block storage family. A container may represent a piece of media (such as a CD-ROM disk), a media holder (such as a CD-ROM cassette), or a physical device (such as a ejectable floppy drive). Containers describe the physical hierarchy of storage devices and media available to Mac OS 8.

container policy An algorithm to manage changes in the accessibility state of a container.

container plug-in A software module that implements a container policy.

control connection A type of connection to a store or a container that allows you to configure the store or container and to change its accessibility state. (A connection to a container is always a control connection.) See also **connection**, **I/O connection.**

derived store A store that maps to another derived store or to a primary store.

extent A variable-length contiguous set of bytes within a store.

I/O connection A type of connection to a store that allows you to read and/or write to the store. See also **connection**, **control connection**.

leaf store A store from which no other stores are derived. Sometimes referred to as a *terminal store*.

mapping (1) The relationship of addresses on two different stores or on a store and a physical device. (2) The process of translating addresses between two stores or between a store and a physical device.

mapping plug-in A software module that translates addresses between a child store and its parent store(s) or between a primary store and a physical device. The mapping plug-in for a primary store is a device driver. Each store has one mapping plug-in.

offset A pointer maintained by the block storage family into a block list descriptor. The value of the offset affects the results of operations performed using that block list descriptor.

parent store Relative to a given I/O request flowing between two stores, the store whose mapping plug-in receives the forwarded request. Compare **child store.**

partition A portion of a device or a store that can be treated as if it were a separate and distinct physical device. Partitions of physical devices are often allocated to a particular operating system, file system, or device driver. Each device or store contains one or more partitions.

partition map 1) Information stored with a device that describes how the device is partitioned into virtual devices. 2) Information stored with a store that describes how the store is partitioned or aggregated into other stores. A physical device always contains a partition map. A virtual device may or may not. A partition map consists of one or more partition map entries.

partition map entry That part of a partition map that describes a single partition, including such things as the starting address of the partition and its length.

partitioning plug-in A software module that creates and maintains a partition map and that makes the information available to the block storage family.

primary store A store that maps to an entity, almost always a physical device, that is beyond the awareness and control of the block storage family.

RAID Redundant array of independent disks.

RAID-5 A type of RAID system.

store A virtual device, an abstraction for a linearly addressable set of any number of blocks. The blocks in a store can reside on a physical device or another store. Each store is associated with one mapping plug-in.

terminal store See **leaf store**.

Device Manager Family

Contents

This chapter describes the Mac OS 8 Device Manager family. The Device Manager family provides a transitional interface for existing device drivers and the applications that interact with them. You should read this chapter if you need to support or maintain System 7 style device drivers.

If you are developing a new device driver you should design it as a plug-in for an existing I/O family, or create your own family if necessary. Drivers written expressly for Mac OS 8 do not need to use the Device Manager.

About the Device Manager Family

The Mac OS 8 Device Manager family is a generic driver service provider that trades performance for compatibility with the System 7 device driver model. Unlike other I/O family services, the Device Manager family is not designed to support any particular driver type. Rather, it provides a compatibility layer for existing drivers and applications that cannot be easily modified or do not require the performance improvements offered by other I/O family services. For example, an expansion card whose driver code resides in ROM, or an application that accesses a driver through the System 7 unit table, can continue to function without modification, but with comparatively lower performance than software that has been redesigned for the Mac OS 8 I/O architecture.

The Device Manager supports generic `'DRVR'` and `'ndrv'` drivers that conform to the restrictions described in this chapter. In general, the Device Manager provides compatibility with the API defined in *Inside Macintosh: Devices* and *Designing PCI Cards and Drivers for Power Macintosh Computers*. However, the I/O architecture of Mac OS 8 imposes restrictions on certain unsupported but previously allowable practices:

- **Resizing parameter blocks or using restricted fields.** Drivers that increase the size of the Device Manager parameter block or use restricted fields must be redesigned to use standard parameter blocks. The `ParamBlockRec` data type is defined in *Inside Macintosh: Devices*.
- **Using private pointers for data transfer operations.** Drivers must use the `PBRead` and `PBWrite` functions to transfer any data that does not fit within the fields of a standard I/O parameter block. For data transfer operations the Device Manager copies or maps only the memory pointed to by the `pb.ioBuffer` field, for the size of `pb.ioReqCount`. Drivers that use the `PBControl` or `PBStatus` functions to transfer data using private

pointers are not supported because the Device Manager cannot ensure the proper memory context across mode changes. The `PBRead` and `PBWrite` functions are described in *Inside Macintosh: Devices.*

Drivers that do not conform to these restrictions must be redesigned, preferably as family plug-ins.

Compatibility with 68K Drivers

Except for the restrictions discussed in this chapter, existing 68K code resources of type `'DRVR'` can operate within the Mac OS 8 cooperative program address space provided

- the driver conforms to the standards defined in *Inside Macintosh: Devices*
- the driver is not a desk accessory
- the driver does not directly interact with hardware devices

Code that interacts directly with hardware devices must be rewritten as a family plug-in or generic native driver. Desk accessories are not supported, and should be redesigned as applications.

Compatibility with Native Drivers

Existing PowerPC native drivers of type `'ndrv'` will operate in the Mac OS 8 environment provided they adhere to the specifications described in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

IMPORTANT

Mac OS 8 allows only one driver per file of type `'ndrv'`. Combining multiple drivers in a single file is not supported. ▲

System 7 style native drivers are installed as generic plug-ins to the Device Manager family and can continue to use the System 7 Device Manager API for communicating with clients in the cooperative program address space. If your driver and it's clients follow the guidelines described in this chapter, they will operate without modification. However, by relying on the Device Manager to provide compatibility, native drivers will experience reduced performance due to the overhead imposed by task wrappers and multiple task switches.

By rewriting your driver and its client applications to use the family and plug-in architecture, you gain more control over I/O processing and performance tuning. If you create your own family you can design the most efficient interface between your plug-in and its clients—bypassing the Device Manager entirely.

Using the Device Manager Family

This section describes how to use the Device Manager to locate, open, and close generic `'ndrv'` plug-ins.

Locating a Generic Plug-In

The following example demonstrates how to use the `DeviceManagerGetDeviceData` function to locate and get information about a generic plug-in. The `MyLookupGenericPlugIn` function returns the plug-in number of a plug-in you specify by name.

```
Boolean MyLookupGenericPlugIn (StringPtr myNdrvName, int *pluginNumber)
{
    OSStatus                    status;
    ItemCount                   returnCount;
    DeviceManagerIteratorData   *dataArray;

    status = DeviceManagerGetDeviceData(0, &returnCount, nil);
    if ( status == noErr ) {
        status = MemNewFixed(xDefaultAllocator,
                        (returnCount * sizeof(DeviceManagerIteratorData)),
                        &dataArray);
        if ( status == noErr ) {
            status = DeviceManagerGetDeviceData(returnCount,
                                                &returnCount,
                                                &dataArray);
            if ( status == noErr ) {
                int index;
                for ( index = 0; index < returnCount ; index ++ ) {
                    if (EqualStrings(
                            dataArray[index].desc.runtimeOptions.driverName,
```

```
                    myNdrvName,
                    true,
                    true)) {
                *pluginNumber = dataArray[index].pluginNumber;
                MemDisposeFixed(xDefaultAllocator,
                        (returnCount * sizeof(DeviceManagerIteratorData)),
                        & dataArray);
                return true; // found the driver
            }
        } // end for
    }
  }
}
return false;
}
```

Opening a Generic Plug-In

The example below shows how you can open a plug-in using the `DoDeviceManagerIO` function.

```
OSStatus MyOpenGenericPlugIn (short *pluginNumber)
{
    ParamBlockRec   pb;
    OSStatus        st;

    BlockZero(&pb, sizeof(pb));
    pb.ioParam.ioNamePtr = kMyDriverName;
    contents.pb = &pb;
    st = DoDeviceManagerIO( nil,
                            contents,
                            kOpenCommand,
                            kImmediateIOCommandKind,
                            nil);
    if ( st == noErr ) {
        *pluginNumber = pb.ioParam.ioRefNum;
    }
    return st;
}
```

Closing a Generic Plug-In

The following example shows how you can close a plug-in using the `DoDeviceManagerIO` function.

```
OSStatus MyCloseGenericPlugIn (short pluginNumber)
{
    ParamBlockRec   pb;
    OSStatus        st;

    BlockZero(&pb, sizeof(pb));
    pb.ioParam.ioRefNum = pluginNumber;
    contents.pb = &pb;
    st = DoDeviceManagerIO( nil,
                            contents,
                            kCloseCommand,
                            kImmediateIOCommandKind,
                            nil);
    return st;
}
```

Device Manager Reference

This section describes the constants, data types, and functions provided by the Device Manager family to support generic native drivers. The data types and functions that provide compatibility with the System 7 Device Manager API are documented in *Inside Macintosh: Devices* and *Designing PCI Cards and Drivers for Power Macintosh Computers*.

Data Types

Command Codes

The Device Manager family defines the following constants for specifying command codes to a driver.

```
/* command codes */
enum {
    kOpenCommand        = 0,                    /* open */
    kCloseCommand       = 1,                    /* close */
    kReadCommand        = 2,                    /* read */
    kWriteCommand       = 3,                    /* write */
    kControlCommand     = 4,                    /* control */
    kStatusCommand      = 5,                    /* status */
    kKillIOCommand      = 6,                    /* kill I/O */
    kInitializeCommand  = 7,                    /* initialize */
    kFinalizeCommand    = 8,                    /* finalize */
    kReplaceCommand     = 9,                    /* replace */
    kSupersededCommand  = 10                    /* superseded */
};
```

See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about how these constants are used by native drivers.

Command Kinds

The Device Manager family defines the following constants for specifying command kinds to a driver.

```
/* command kinds */
enum {
    kSynchronousIOCommandKind   = 0x00000001,   /* synchronous */
    kAsynchronousIOCommandKind  = 0x00000002,   /* asynchronous */
    kImmediateIOCommandKind     = 0x00000004    /* immediate */
};
```

See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about how these constants are used by native drivers.

Device Manager Family Iterator Structure

The `DeviceManagerGetDeviceData` function returns information about a selected plug-in in the `DeviceManagerIOIteratorData` structure.

```
struct  DeviceManagerIOIteratorData {
        IOCommonInfo        ioCI;
        short               pluginNumber;
        short               refNum;
        DriverDescription   desc;
};
typedef struct DeviceManagerIOIteratorData DeviceManagerIOIteratorData;
```

`ioCI` A structure of type `IOCommonInfo`. It contains the device reference number from the Name Registry, and the version number of the family's iterator structure. For more information about the `IOCommonInfo` structure, see "About the I/O Architecture" (page 1-3)

`pluginNumber` A unique identifier assigned by the Device Manager to each generic plug-in. You use this identifier to specify a plug-in when using the `DoDeviceManagerIO` function. Plug-in numbers are in the range 1 to 128.

`refNum` A unique identifier used by applications in the cooperative program address space for identifying a driver in the unit table. Mac OS 8-savvy applications should use the `pluginNumber` parameter and set `refNum` to 0. See *Inside Macintosh: Devices* for information about driver reference numbers and the unit table.

`desc` The driver description exported by the driver. The `DriverDescription` structure is used to compare a driver's functionality with a device's needs. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for more information about this structure.

I/O Command Contents Structure

You pass commands to the `DoDeviceManagerIO` function in a structure of type `IOCommandContents`. This structure encapsulates the System 7 Device Manager parameter block along with additional information for native drivers.

```
union IOCommandContents {
    ParmBlkPtr              pb;
    DriverInitInfoPtr       initialInfo;
    DriverFinalInfoPtr      finalInfo;
    DriverReplaceInfoPtr    replaceInfo;
    DriverSupersededInfoPtr supersededInfo;
};
typedef union IOCommandContents IOCommandContents;
```

`ParmBlkPtr` A pointer to a Device Manager parameter block (a structure of type `ParamBlockRec`). See *Inside Macintosh: Devices* for information about this structure.

`DriverInitInfoPtr`
A pointer to a `DriverInitInfo` structure. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about this structure.

`DriverFinalInfoPtr`
A pointer to a `DriverFinalInfo` structure. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about this structure.

`DriverReplaceInfoPtr`
A pointer to a `DriverReplaceInfo` structure. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about this structure.

`DriverSupersededInfoPtr`
A pointer to a `DriverSupersededInfo` structure. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about this structure.

Functions

DeviceManagerGetDeviceData

Returns information about generic `'ndrv'` plug-ins.

```
OSStatus DeviceManagerGetDeviceData(
                    ItemCount requestItemCount,
                    ItemCount *totalItemCount,
                    DeviceManagerIOIteratorData **dataArray);
```

`requestItemCount`
The number of `DeviceManagerIOIteratorData` structures you have allocated to the `dataArray` parameter. Set to 0 if you do not want the function to return information in the `dataArray` parameter.

`totalItemCount`
A pointer to an item count. On output, the number of installed plug-ins reported by the Device Manager.

`dataArray` A pointer to an area of memory to hold an array of `DeviceManagerIOIteratorData` structures. You must allocate enough memory to hold the requested number of structures. On output, the Device Manager returns the requested number of `DeviceManagerIOIteratorData` structures. Set to `nil` if you do not want the function to return this information.

function result A result code.

DISCUSSION

This function returns information maintained by the Device Manager for each installed plug-in. You request the information by providing an array of Device Manager family iterator structures, and the function returns the latest information in this array.

The `totalItemCount` parameter returns the number of installed plug-ins. You can use this value to determine how much memory to allocate for the `dataArray` parameter. If `requestItemCount` is 0 and `dataArray` is `nil`, the function

returns the number of installed plug-ins without returning information about them.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

DoDeviceManagerIO

Sends a command to a generic `'ndrv'` plug-in.

```
OSStatus DoDeviceManagerIO(
                    IOCommandID commandID,
                    IOCommandContents contents,
                    IOCommandCode code,
                    IOCommandKind kind,
                    KernelNotification *notify);
```

`commandID` Reserved for Device Manager internal use. Set to `nil`.

`contents` A pointer to an `IOCommandContents` data structure. See "I/O Command Contents Structure" (page 8-10) for the definition of the `IOCommandContents` data structure.

`code` A command code constant, such as `kOpenCommand` or `kCloseCommand`, that describes the type of action to be performed. See the "Command Codes" (page 8-7) section for the definition of the command code constants.

`kind`	A command kind constant, such as `kAsynchronousIOCommandKind`, that specifies how the Device Manager should process the request. See the "Command Kinds" (page 8-8) section for the definition of the command kind constants.
`notify`	A pointer to a microkernel notification structure you create to indicate how you want your application to be notified when the I/O command is completed. The Device Manager uses the specified notification method when the plug-in calls the `IOCommandIsComplete` function. This field is required for asynchronous I/O commands, and ignored for others. See *Inside Macintosh: Microkernel and Core System Services* for information about microkernel notification.
function result	A result code.

DISCUSSION

The `DoDeviceManagerIO` function sends commands and I/O requests to a plug-in. The `contents`, `code`, and `kind` parameters are passed directly to the plug-in's `DoDriverIO` entry point. See *Designing PCI Cards and Drivers for Power Macintosh Computers* for information about the `DoDriverIO` entry point.

There are two methods for identifying the plug-in you are sending a request to. Applications in the cooperative program address space can use the reference number derived from the driver's position in the unit table, a value in the range -1 to -128. Mac OS 8-savvy applications will use the plug-in number returned by the `DeviceManagerGetDeviceData` function, a value in the range 1 to 128. In either case, the value is stored in the `pb.ioRefNum` field of the `IOCommandContents` structure.

If the `kind` parameter is set to `kImmediateIOCommandKind`, the Device Manager passes the command directly to the plug-in, bypassing the request queue and without causing a task switch. Other kinds of commands are queued for later

processing by the plug-in. Table 8-1 shows the command kinds that a client may specify for a given command code.

Table 8-1 Supported I/O command kinds and command codes for clients

	Immediate	Synchronous	Asynchronous
`kOpenCommand`	•	•	
`kCloseCommand`	•	•	
`kReadCommand`	•	•	•
`kWriteCommand`	•	•	•
`kControlCommand`	•	•	•
`kStatusCommand`	•	•	•
`kKillIOCommand`	•	•	
`kInitializeCommand`			
`kFinalizeCommand`			
`kReplaceCommand`	•	•	
`kSupersededCommand`	•	•	

The `kInitializeCommand` and `kFinalizeCommand` codes are issued only by the Device Manager, and cannot be used by clients. If you are developing a plug-in, it must be able to accept the command kinds and codes listed in Table 8-1, even if it cannot process them in the requested manner. Your plug-in must return an appropriate error result code if it cannot process the request as specified. In addition, your plug-in must support immediate requests from the Device Manager for the `kInitializeCommand` and `kFinalizeCommand` codes.

As documented in *Designing PCI Cards and Drivers for Power Macintosh Computers*, your driver should complete immediate I/O requests using a normal function return, not by calling the `IOCommandIsComplete` function. Synchronous and asynchronous requests must return through `IOCommandIsComplete`.

See *Designing PCI Cards and Drivers for Power Macintosh Computers* for detailed information about writing generic native drivers.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Booting Services

Contents

Booting, also called bootstrapping, is the process by which a computer system initializes and configures itself after its power is turned on. This chapter describes the Mac OS 8 booting sequence and the software components that provide those services.

About Mac OS 8 Booting Services

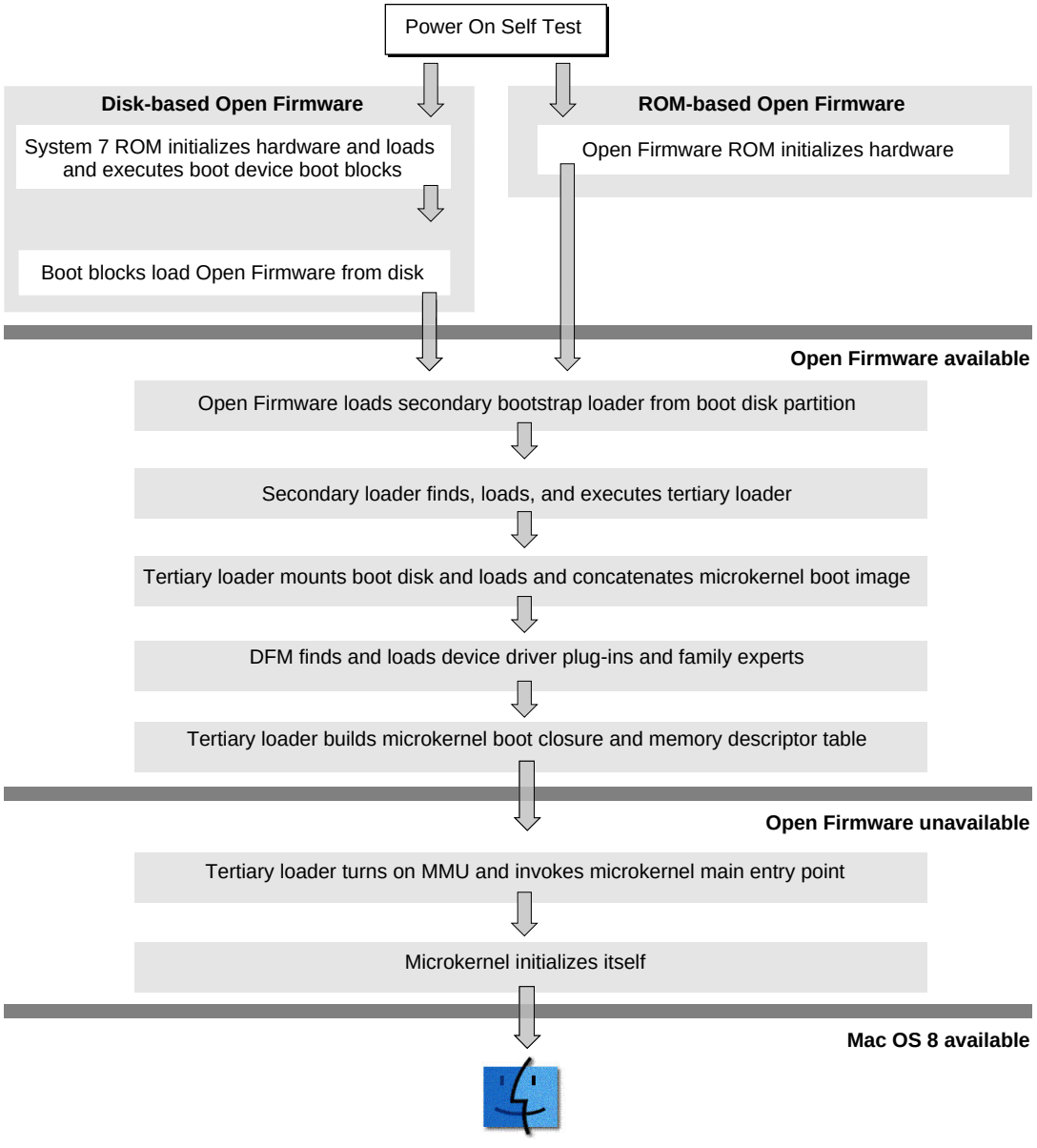
The booting subsystem prepares the environment in which the microkernel is to execute, loads the microkernel and its ancillary system-specific software, and transfers control to the microkernel's main entry point.

Booting services in Mac OS 8 are provided by the following software components

- boot blocks (on the booting media, typically a hard disk or CD-ROM)
- primary bootstrap loader (either ROM-based Open Firmware or System 7 ROMs with a disk-based Open Firmware)
- secondary bootstrap loader (from the boot media or network connection)
- tertiary bootstrap loader (in the Mac OS folder)

The Mac OS 8 booting process can proceed along either of two paths: one for machines containing Open Firmware ROMs, and another for machines containing only System 7 ROMs. Open Firmware is the primary loader for machines containing Open Firmware ROMs. The primary loader for machines containing only System 7 ROMs is a disk-based Open Firmware implementation loaded by modified System 7 boot blocks. The two paths converge at the secondary loader; from that point forward, the booting code is the same for all platforms (except for plug-ins to support hardware variations). Figure 9-1 illustrates the Mac OS 8 booting sequence.

Figure 9-1 Mac OS 8 booting sequence



Booting Sequence

This section describes the sequence of operations that execute to complete the Mac OS 8 booting process. The initial steps vary, depending on whether or not the machine contains Open Firmware ROMs.

Hardware Self-Test

The first action of the system is always a system hardware self-test, which is also called a Power-on Self-Test, or POST. The POST tests the system's RAM configuration and determines the amount of RAM available. At this time the system determines the presence or absence of optional hardware devices. As a result of the hardware self-test, the system firmware—code residing in ROM—has knowledge of the basic hardware present in the machine.

ROMs and Boot Blocks

Mac OS 8 is designed to run on machines that boot from either Mac OS System 7 ROMs or ROM-based Open Firmware (booting code conforming to the IEEE standard 1275-1994).

On machines containing only System 7 ROMs, the system's first action following the hardware self-test is to load the boot blocks from the boot volume and begin their execution. The boot blocks determine whether the machine is configured to run Mac OS 8 or some version of System 7. If the system software is Mac OS 8, the boot blocks load a disk-based copy of Open Firmware from a file in the Mac OS folder and begin its execution.

Once Open Firmware is running, whether it resides in ROM or was loaded into RAM by the modified System 7 boot blocks, the Mac OS 8 booting sequence is virtually identical for either system.

Open Firmware

As soon as Open Firmware begins running, the specified Open Firmware booting sequence begins. Open Firmware determines the boot device by examining the `/options` device tree system node, which is initialized at startup

from data stored in nonvolatile RAM. This node contains properties that define the boot device and the boot operation's optional parameters. It also defines the properties specifying the device (or devices) through which the Open Firmware user interface should perform I/O.

Open Firmware then executes the boot-command string which finds the first secondary loader partition on the boot disk, loads its contents into memory, and jumps to its entry point.

Secondary Loader

The secondary loader can be loaded from an easily found partition present on the boot media or in a file read through a network connection. The secondary loader uses its rudimentary file system support to load and execute the tertiary loader.

As the Mac OS 8 secondary loader begins execution, one or more I/O function wrappers can optionally be invoked to execute in sequence. These function wrappers are found initially by the first stage of the secondary loader. After any such function wrappers finish execution, the final Mac OS 8 booting subsystem's secondary loader continues. The secondary loader's main purpose is to find and load the tertiary loader.

Once the secondary loader has begun executing, machine-specific and ROM-specific code can be accessed through the Open Firmware client interface. Then the secondary and tertiary loaders can make machine-specific calls to perform I/O and determine hardware configuration.

Tertiary Loader

The tertiary loader contains the remainder of the booting software. It loads two kinds of software. First, the tertiary loader loads the microkernel and its required dynamically linked code fragments (such as the file system and Name Registry). Second, the tertiary loader loads I/O family experts and appropriate device plug-ins, providing access to the boot media and basic user-interaction facilities.

The tertiary loader identifies within the Mac OS folder hierarchy a collection of files required in the microkernel's boot image. Each of these files has a filetype of `'maxb'` and contains one or more PEF (preferred executable format) code fragments. These fragments contain the microkernel itself and needed subsystems, such as the file system and Name Registry. For each fragment, the

tertiary loader uses the Open Firmware client interface to allocate memory and loads the fragment.

Among the code fragments comprising the microkernel boot image, only the microkernel's initialization plug-in has a filetype of `'krnl'` and a usage field stating that its fragment type is an application (`kIsApp`). The tertiary loader invokes this processor-dependent fragment's entry point when it has finished preparing the microkernel for execution. The tertiary loader also takes special care to align the first byte of this fragment's code section to a page boundary so that the PowerPC alignment exception handler can function properly.

The tertiary loader includes a subset of the driver and family matching (DFM) service to locate software needed to access I/O devices required during booting. The DFM service walks the Open Firmware device tree and, for each device node, locates the plug-ins and family experts associated with that device, unless previously loaded from ROM by Open Firmware. For each device, the DFM service creates an entry in the Name Registry with properties for a pointer to its plug-in and its plug-in description data structure. The DFM service then loads the family expert associated with each required device, and the family expert selects and loads the most suitable driver plug-in for each device. Boot devices include those for the boot disk, real-time clock, primary display, nonvolatile RAM, and mouse and keyboard. The mouse and keyboard devices are for user interaction if a problem occurs during booting. See "Driver and Family Matching Service" (page 9-11) for more information about the DFM service.

The tertiary loader then identifies the low-level expert for the motherboard of the running system and places a pointer to it on the `driver-reg,AAPL,MacOS,PowerPC` property of the device tree root node.

At this point, the tertiary loader invokes its built-in, boot-time version of the Code Fragment Manager to build the microkernel boot closure. This process includes allocating and setting up expanded static data areas required by each code fragment in the closure and performing relocations required for the fragments to execute in their final base addresses.

The tertiary loader then builds a data structure describing the memory areas in the system to facilitate initialization of the microkernel's memory management tables. The tertiary loader builds an exportable representation of the Open Firmware device tree. All code and data structures passed into the microkernel are then copied to their proper kernel runtime logical addresses, so that the microkernel's memory address mappings will be correct.

Additional details of tertiary loader facilities are provided in "Booting Services Software" (page 9-8).

Invoking the Microkernel

Finally, the tertiary loader invokes the microkernel at its main entry point, passing in parameters containing all the boot-time information it previously constructed. The microkernel's processor-dependent code module then prepares its memory management tables and initializes the processor's memory management unit (MMU). The MMU performs the address translations defined in the memory area description tables passed in from the tertiary loader. The microkernel then initializes its remaining subsystems so that other components of the system can use its entire API as they initialize themselves.

When the microkernel is ready to run the rest of the system, the motherboard expert is prepared and executed from its location in the `driver-reg,APPL,MacOS,PowerPC` property of the device tree's root node. The motherboard expert works with the DFM service to initialize any I/O device plug-ins and low-level experts required to inform the user of the booting sequence progress and to perform the I/O operations required to get the file system running.

Booting Services Software

This section presents additional details of certain components of the Mac OS 8 booting subsystem.

Boot Blocks

Boot blocks are the first two 512-byte blocks on a bootable HFS-formatted volume. System 7 ROMs execute the code contained in the boot blocks of the boot volume, specified in the machine's nonvolatile RAM, as soon as they complete their initial power-on tests. System 7 boot blocks contain less than 1024 bytes of 68K code and some localization and configuration parameter data. This code opens the System File and executes the contents of one of its resources.

To boot Mac OS 8 system efficiently on machines with System 7 ROMs, the System 7 booting sequence must be interrupted very early, and the first opportunity to do so is afforded by the boot blocks. Mac OS 8 provides modified boot blocks that introduce the Mac OS 8 booting sequence when they execute. The Mac OS 8 boot blocks contain additional 68K code and data, which are necessary to permit the system to determine the user's preference for Mac OS 8 and to load a disk-based version of Open Firmware.

Disk-Based Open Firmware

During the Mac OS 8 booting sequence, as executed on a machine with Mac OS System 7 ROMs, modified boot blocks load and begin execution of a disk-based version of Open Firmware. This disk-based Open Firmware (DBOF) implements the entire IEEE standard 1275-1994 for the System 7 CPU platform.

When it begins to execute, the DBOF assumes control of the machine precisely as if it were running from the Mac OS 8 boot ROM. DBOF supports all devices supported by the Open Firmware standard and any additional devices for which Open Firmware Forth or Mac OS 8-style drivers have been constructed. All main logic board devices that must participate in the booting sequence must have trivial device drivers built into DBOF in the form of Forth code.

Note
Subsequent releases of Mac OS 8 will support a facility for loading boot-time drivers using System 7's file system to support booting from devices that contain no Open Firmware ROM code.

Embedded HFS Package

The embedded HFS package, which is part of the tertiary loader, contains a complete, read-only implementation of the Mac OS hierarchical file system (HFS), including extended support for volumes exceeding 4 GB. This implementation includes the ability to mount HFS-formatted disk volumes, determine the Mac OS folder's location on the volume, enumerate the contents of folders relative to any fixed point, obtain file type and creator (Finder) information, and read information from both the data and resource forks of any files found.

Embedded Resource Manager

The embedded resource manager, which is part of the tertiary loader, implements a read-only Mac OS resource extraction facility. The Mac OS 8 booting sequence uses this capability to extract localizable strings, graphics, and `'cfrg'` resources from the resource forks of embedded HFS files.

Self PEF Loader

The self PEF loader, which is part of the tertiary loader, relocates in place a PEF (preferred executable format) container into memory and prepares it for execution. A PEF container is a storage object encompassing a code fragment in Apple's standard format for use by the Code Fragment Manager.

The function of the self PEF loader in the booting sequence is to prepare the tertiary loader for execution. The self PEF loader expands in place all static data, performs relocations of data references to relocatable code and data objects, and identifies the TOC and entry point of the PEF container to its caller. Because the tertiary loader forms a complete closure with no imports, simply loading its PEF container and preparing its static data, including its TOC, is sufficient to prepare it for execution.

Boot-time Code Fragment Manager

The boot-time code fragment manager (CFM) is part of the tertiary bootstrap loader. The boot-time CFM resolves all the import and export interdependencies among the code fragments comprising the microkernel boot image, and it relocates any references to addresses that are contained in the static data areas of the fragments. For example, the boot-time CFM relocates the pointers in a fragment's table of contents (TOC) to refer properly to their final address ranges. The boot-time CFM supports only code fragments resident in memory.

The boot-time CFM includes a PEF module transitive closure calculator. This facility determines the complete set of all libraries imported by a root PEF container, including additional libraries to which the imported libraries refer. That is, all of the code fragments in the microkernel boot image must be able to bind to each import library referred to in their TOCs. To do so, the addresses of the referents of each code fragment's imports must be placed in its TOC. This software finds the set of all import libraries needed to prepare every PEF

container required by the boot image. Each of these import libraries must be present in the boot image or the device tree to satisfy all import requirements.

Device Tree Maintenance Facility

The device tree maintenance facility is used by the tertiary loader to transform the contents of the Open Firmware client interface's device tree. The facility puts the data into a compact form that can be conveniently passed as a parameter by the tertiary loader when it invokes the microkernel. To accomplish this, the structure of the data is copied into an exportable form which is not position-dependent. This data structure is used subsequently by the driver and family matching service.

Driver and Family Matching Service

The tertiary loader contains a subset of the Mac OS 8 driver and family matching (DFM) service, which locates all the plug-ins and family experts required to boot the machine. First, the DFM service traverses the position-independent Open Firmware device tree and copies the data into the actual Name Registry device subtree using the Name Registry APIs. For the required name entries in the device portion of the Name Registry, the DFM service locates, loads, and instantiates the plug-in and family expert best suited to support the given device. The booting subsystem does not require disk-based plug-ins for devices already exporting ROM-based plug-ins.

See Chapter 2, "Driver and Family Matching," for more information about the driver and family matching service.

Index

A

B

D

E

I

K

M

P

R

S