Developer Note

Developing PC Card Software for the Mac OS

Developer Press

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

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

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

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

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

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

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

Contents

Figures and Tables

About This Developer Note

This developer note describes how the Personal Computer Memory Card International Association (PCMCIA) expansion card interface is implemented in PowerBook computers. The term PC Card is used throughout this note to indicate expansion cards defined by the PCMCIA standard.

Apple provides full software support for PC Cards, including

- close adherence to the PCMCIA standard
- seamless integration into the Macintosh platform and user experience
- a high level of compatibility with existing and future PC Cards

This note is written for professional hardware and software engineers who are generally familiar with existing Macintosh technology and have previously read the PCMCIA standard. If you would like more information about the PCMCIA standard and about Macintosh technology, see the documents listed in "Supplementary Documents" beginning on page xi.

Contents of This Note

This note is divided into five chapters:

- Chapter 1, "Overview," introduces the general features and concepts of the PowerBook system software that supports PC Cards.
- Chapter 2, "Client Software," describes how to write client software for the Card Services application programming interface (API).
- Chapter 3, "Card Services Routines," describes the Card Services portion of the PC Card Manager, which constitutes the primary Macintosh system software support for PC Cards in PowerBook computers.
- Chapter 4, "Device Drivers," provides guidelines for developers writing device drivers compatible with PowerBook computers.
- Chapter 5, "Human Interface," describes the installation and operation of PC Cards from the user's viewpoint and provides human interface guidelines for developers of PC Card software.

At the end of this book are a glossary and an index.

Conventions and Abbreviations

This developer note uses the following typographical conventions and abbreviations.

Typographical Conventions

Terms that appear in the Glossary, are shown in **boldface** where they are first appear in the main body of text.

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

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

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

▲ **WARNING**
A note like this indicates a potential problem that could damage hardware, cause the software to crash, or cause permanent data loss. ▲

Abbreviations

Abbreviated units of measurement used in this note include

KB	kilobytes	MHz	megahertz
MB	megabytes	V	volts

Other abbreviations used in this book include

API	application programming interface
CIS	Card Information Structure
DCE	device control entry
EEPROM	electrically-erasable programmable ROM
HFS	hierarchical file system
JEDEC	Joint Electron Device Engineering Council
MTD	Memory Technology Driver
NVRAM	nonvolatile RAM
PCMCIA	Personal Computer Memory Card International Association

PDS	processor-direct slot
RAM	random-access memory
ROM	read-only memory
SRAM	static RAM
UPP	universal procedure pointer

Supplementary Documents

This section describes technical documents that supplement the material in this book.

PCMCIA Documents

There are two primary sources of information about PCMCIA standards.

- The first document is *PCMCIA Standards*, Standard Release 2.01—November 1992. Current Apple hardware and software supports this release, and you should read the following sections of this book if you want to develop client software for PowerBook computers: Card Services Specification, Socket Services Specification.
- The latest version of the document is *PC Card Standard*, February 1995. This book contains the same information as the first document, but it also contains additional information on standards developed since 1992. The document consists of a number of volumes, and the ones most relevant in this context are: Volume 1, Overview and Glossary; Volume 5, Card Services Specification; and Volume 6, Socket Services Specification.

To simplify references to these documents, if you can use either book, you are referred to *PCMCIA Standards*. If the information you need is only in the latest version, you are referred to *PC Card Standard*, or the February Release. If the reference is to the actual standard, it is referenced as PCMCIA standard.

Both books are published by the Personal Computer Memory Card International Association, and you can order them from

Personal Computer Memory Card International Association
1030G East Duane Avenue
Sunnyvale, CA 94086
Phone: 408-720-0107
Fax: 408-720-9416

Apple Documents

Apple Developer Press publishes a variety of books and technical notes designed to help third-party developers design hardware and software products compatible with Apple computers.

- *Inside Macintosh* is a collection of books, organized by topic, that describe the system software of Macintosh computers. Together, these books provide the essential reference for programmers, software designers, and engineers. *Designing Cards and Drivers for the Macintosh Family,* third edition, explains the general software requirements for drivers compatible with Macintosh computers.
- *Technical Introduction to the Macintosh Family,* second edition, surveys the complete Macintosh family of computers from the developer's point of view.
- *Macintosh Human Interface Guidelines* provides authoritative information on the theory behind the Macintosh "look and feel" and Apple's standard ways of using individual interface components. A companion CD-ROM disk, *Making It Macintosh,* illustrates the Macintosh human interface guidelines through interactive, animated examples.

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

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

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

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

CHAPTER 1

Overview

Overview

This chapter gives an overview of the software architecture that supports PC Cards installed in PowerBook computers. It includes overviews of:

- The PCMCIA standard developed by the Personal Computer Memory Card International Association.
- Elements of the architecture, including Card Services, Socket Services, and drivers as clients. Card Services and Socket Services are the elements of the architecture that conform to the PCMCIA standard. Chapter 3 of this developer note provides detailed information about Socket Services. The note does not deal in detail with Socket Services. However, you will find comprehensive information on the subject in *PCMCIA Standards*, published by the Personal Computer Memory Card International Association, and referred to throughout this developer note as *PCMCIA Standards*.
- The programming model.
- PowerBook implementation of the PCMCIA standard.

The architecture includes a PC Card Manager and client software written by Apple Computer, Inc. and by third-party developers. Apple supplies the system software for PC Cards in the ROMs of certain PowerBook computers. For details about the hardware support for PC Cards, refer to the developer documentation provided with specific PowerBook models.

Overview of PCMCIA Standards

The goal of the PCMCIA is to promote the interchangeability of Integrated Circuit Cards (IC cards) among various computers and electronic products. The cards are referred to in this developer note as PC Cards, and they are 68-pin I/O cards that provide:

- memory storage
- fax/modem implementation
- local area networks (LANs) implementation
- video support

There are three types of PC Cards: Type I, Type II, and Type III. The cards are 3.370″ by 2.126″, but differ in thickness, with the Type III card being the thickest. Type I and Type II cards can be accommodated in the same type of slot. Type III cards requires a deeper slot. In computers, such as the PowerBook, that provide two stacked slots, Type III cards are usually plugged into the lower slot, but occupy the physical space of both slots. Computers with two stacked slots can therefore accommodate: two Type I cards, two Type II cards, one Type I card and one Type II card, or one Type III card. The standards for PC Cards are defined in *PCMCIA Standards*.

Mechanical Considerations for PC Card Developers

The Apple PCMCIA card slot is a unique design, that incorporates a software eject capability. As a PC Card is inserted into the slot, the card presses against a spring-loaded lever and then latches into place. The stored energy in the spring is used later to eject the card. As a developer of PC Cards, you have some design choices that are recommended to avoid mechanical incompatibilities with the Apple card slot.

Optimal Square Corner Design

When a PC Card is inserted into the slot, the leading edge of the card, which has a female connector, is used to press against the spring-loaded lever. The process works most effectively if your PC Card has square corners on the leading edge. Although rounded corners will work, it is recommended that the radius of these corners be less than 2 mm. Apple recommends that your card be designed with square corners on the leading edge.

Type III Cards

Type III cards are built so that they have essentially a double thickness. The PCMCIA mechanical specification is very specific on the dimensions of the lower part of the Type III card, but is very vague about the shape of the upper part of the card.

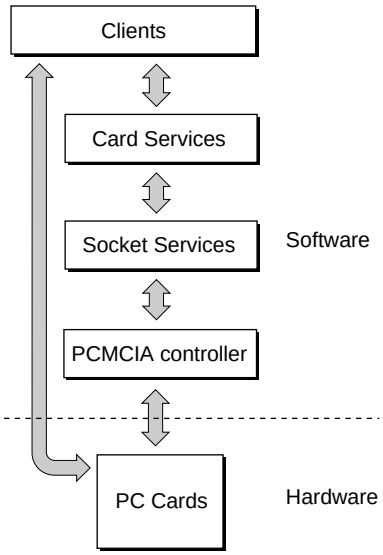
When a Type III card is inserted into the Apple card lower slot, the upper half of the card actually presses against the ejection lever for the upper slot. Apple recommends that you design the mechanical outline of the Type III card in such a way that the upper half of the card cannot snag or bind on the upper arm of the card slot.

Since the PCMCIA specification allows much freedom in the shape of the upper half of the card, Apple recommends that you test a prototype of your card in the Apple card slot to ensure smooth insertion and ejection.

Overview of the Software Architecture

The PowerBook software support for PC Cards follows the traditional layered architectures seen in the Macintosh platform, with Card Services and Socket Services comprising the operating system portion. Unlike other Card and Socket Services implementations, the PowerBook implementation does not allow clients access to the Socket Services layer. Figure 1-1 gives an overview of the software architecture that supports PC Cards in PowerBook computers.

Figure 1-1 Software architecture for PC Card support



- **Clients** are device drivers or application software that use Card Services.
- **Card Services** is the operating system layer that supports PC Card sockets and Socket Services software. Card Services software also provides resource management for clients of PC Cards. Table 1-1 lists some events that Card Services reports to clients. It illustrates the relationship between the support architecture and its clients. Card Services routines are described in detail in Chapter 3 of this developer note.
- **Socket Services** is the operating system layer that provides the upper layers of software with **hardware abstraction** from socket controllers and adapters. PowerBook support for PC Cards contains both Card Services and Socket Services software layers. This developer note does not provide detailed information about Socket Services. For this type of information, you should refer to *PCMCIA Standards*.
- The **PCMCIA controller** is the hardware interface to the PC Cards. It provides the interface signals, configurable voltages to power the cards, hardware windows into the card's address space, and interrupt decoding for state changes.
- **PC Cards** themselves contain the hardware interface to the PowerBook's PCMCIA bus, as well as the hardware required to implement the card's function (memory, fax/modem, local area networks, and so on). The cards may also have a **Card Information Structure (CIS).** This is a list of structures that describe the card's functions and capabilities.
- **Sockets** (not shown in Figure 1-1) are the actual hardware receptacles that accept PC Cards. The PowerBook hardware implementation supports two stacked sockets that allow the user to insert two Type I or Type II cards, or one Type III card. When the Type III card is installed, its interface connector occupies only one socket (usually the lower socket). However, because the card is thicker than Type I and Type II cards, it occupies both slots.

Table 1-1 Sample of events reported by Card Services to clients

Event message	Meaning
`kCSBatteryDeadMessage`	The PC Card battery is no longer serviceable and data may be lost.
`kCSBatteryLowMessage`	The PC Card battery is weak and should be replaced. Data integrity of the PC Card is maintained.
`kCSCardReadyMessage`	The PC Card's +RDY/–BSY line has transitioned from the busy to the ready state.
`kCSCardRemovalMessage`	The PC Card has been removed from its socket.
`kCSCardInsertionMessage`	The PC Card has been inserted into a socket, or a client has just registered for insertion events. Card Services is creating artificial insertion events for the PC Cards that are already in sockets.
`kCSFunctionInterruptMessage`	The PC Card's interrupt request (–IREQ) line has been asserted.

Card Services

Card Services supports multiple clients and multiple Socket Services modules. Card Services provides client registration, resource management, memory services, client utilities, and advanced client utilities. The PowerBook Card Services architecture coordinates access to sockets (through the Socket Services software interfaces) and access to system resources. There is only one executing copy of Card Services in the host system.

Installation

The Card Services software is loaded from ROM and requires Macintosh System 7 or later. It installs a trap for opcode $AAF0 and registers with the Gestalt Manager to let other software know that it is installed. The Gestalt selector (`pccd`) determines whether Card Services is installed. See "PC Card Manager Constants" beginning on page 68 for the Gestalt attribute definitions.

Card Services performs the following installation processes:

- Accesses the socket hardware through Socket Services software, described in "Socket Services" beginning on page 7.
- Receives all interrupt notification of socket changes from Socket Services.
- Passes status changes, interrupt notification, and other messages to clients through a standard callback architecture. The clients of Card Services register with Card Services when they need to communicate with PC Cards. At registration time the clients pass a callback handler address to Card Services and an event mask to remove unwanted events for a particular socket.

- Numbers resources (adapters, sockets, windows, pages, and so on) as they are registered. Most numbering is zero-based in Card Services. For example, if two sockets are registered by an adapter and they are the first two sockets to be registered with Card Services, Card Services will number them socket 0 and socket 1. The next set will be numbered 2, 3, and so on. Any undefined fields in the Card Services interface definitions should be set to 0 for compatibility with future Card Services revisions.

Operation

Events and the subsequent callbacks from Card Services to clients are generated from a variety of status conditions and card interrupts, including phantom events that Card Services manufactures for clients. Table 1-1 on page 5 lists some of the system messages. See "PC Card Manager Constants" beginning on page 68 for a complete list of Card Services callback events. Card Services itself is a state machine that waits for clients and Socket Services modules to register with it. Card Services is driven by the actions of a client or the actions of a Socket Services module.

Card Services performs the following operating tasks:

- Provides the means to register PC Card clients.
- Prioritizes and dispatches a card event, interrupt, or status change notification back to registered clients. The PCMCIA standard specifies how callbacks to card clients are prioritized.
- Provides the minimum CIS parsing needed to recognize and provide support for different types of memory devices, and to provide simple ways for clients to extract information from the CIS when they do not have the knowledge to do this.
- Provides the means to configure a PC Card in a specific socket.
- Provides the means to register Socket Services modules that support other types of PCMCIA adapters.
- Provides OS (operating system) services to Socket Services modules. Such services include routing interrupt callbacks, static (global) data space allocation and deallocation, isolation from virtual memory requirements, and so on.

Card Services returns multibyte fields in **little-endian** format, which is the way most PC Cards store data. With little-endian addressing, the address for a field refers to its least significant byte, as opposed to **big-endian** addressing, where the address for a field refers to its most significant byte. Macintosh computers use big-endian addressing.

Card Services supports the conversion of the little-endian to the big-endian format for multibyte data. The conversion is done in the Socket Services layer, usually within the controller itself. PowerBook Card Services lets client software control the addressing format of multibyte data.

Card Services is **reentrant** by design. However, most of the functions that Card Services provides are **synchronous** and control the configuration of PC Cards, not the interactions with the functions that are on the PC Card. Only a few of the many Card Services functions are designed to be **asynchronous.** Once a card is configured by a client, the client usually accesses the card registers or card memory directly. *PCMCIA Standards* describes in detail the nature of each function call.

Socket Services

Beneath the Card Services layer lies one or more Socket Services modules, each of which is responsible for providing a common API (application programming interface) for Card Services to call, and for routing communication between Card Services and the socket controller hardware. Each Socket Services module is tailored to a specific piece of socket controller hardware, which is either on the main logic board or on a PCMCIA adapter.

Socket Services is part of the hardware abstraction layer. It presents a standard API to Card Services that is unaffected by socket hardware changes. Socket Services is responsible for handling the interrupt generation and interrupt control processes of its particular hardware. When an interrupt is generated by hardware, Socket Services accepts the interrupt and calls the Card Services entry point with information about the socket and adapter that caused the interrupt and the nature of the interrupt.

A Socket Services module is specific to a PCMCIA controller. The initial Macintosh implementation includes a Socket Services module for the hardware controller. Each Socket Services module owns one or more host adapters, and each host adapter may have multiple PCMCIA sockets associated with it.

Socket Services modules are responsible for

- Registering and acknowledging interrupt generation from the adapter
- Validating access parameters for adapters within its scope of control
- Relaying all control calls from Card Services to the adapter
- Informing Card Services of the Socket Services capabilities and attributes

The `CSAddSocketServices` routine is called by a Socket Services module during initialization to inform Card Services of its presence and attributes. Various entry points and details of the Socket Services are passed to Card Services in the `AddSocketServicesPB` parameter block. The last field in `AddSocketServicesPB` is a pointer that is passed to Socket Services each time it is called by Card Services. In this way Socket Service modules can retain global variables or other data that is dependent upon the implementation.

Drivers as Clients

The support software for the PowerBook PC Card provides the functionality that PC Card client developers require, using an architecture that resembles the traditional Macintosh environment. The Card Services and Socket Services APIs are very similar to the APIs presented in the PCMCIA standard. The PowerBook architecture also includes guidelines for helping client developers implement client loading, PC Card parsing, client event handling, and other aspects of the client environment.

Macintosh drivers are the main clients of Card Services. However, applications, as well as other types of code, can register with Card Services, and can become a client if they wish. You should note the following client issues:

- *Client loading.* Whether it is loaded from a resource of type '`INIT`', from a configuration ROM, or from an application program, the PowerBook architecture defines fast, efficient ways for a client to inspect PC Card resources and to determine whether it is appropriate for a client to control a particular PC Card.
- *Client interrupt and message handling.* Clients of PC Cards receive all event, interrupt, and status change notifications through a callback mechanism. Clients do not need to register interrupt handling routines specific to an adapter because that is taken care of by Card Services and Socket Services. The client must provide an interrupt handling callback address and adhere to the interrupt execution limitations of Macintosh programming. For example, during interrupt time the client must not move memory.
- *Client human-interface responsibilities.* The Macintosh interface demands that PC Cards be tightly integrated into the desktop metaphor. Drivers must provide services that will allow users to manipulate the PC Cards in sensible ways (for example, when ejecting a PC Card). The system determines the events and user actions of which a client may have to be aware, and decides how the client should react.

IMPORTANT

Clients should not bypass the Card Services API to configure PC Cards. Clients that do so may cause synchronization errors within the adapter and eventually cause the system to malfunction. ▲

PC Card clients register with Card Services using a call `CSRegisterClient` routine. Most clients of Card Services will be drivers, which normally load with system extensions. Applications can be clients as well, and they register during initialization. To receive events, a registered client must enable events by calling a `CSVendorSpecific` routine with the function code `vsEnableSockets`.

Arguments to the CSRegisterClient include information about whether the client is a memory client or an I/O client, and also contain the address of an **event handler.** Card Services uses the event handler to notify clients of events, including interrupts from cards. All events generated and delivered to clients use the callback mechanism. Clients must preserve the contents of the arguments used in the callback mechanism, so that when subsequent clients are notified by Card Services, they will see the same arguments in their callback handler. Clients can specify the sockets and event types for which they need to receive callback events. Clients use `CSRequestSocketMask`, `CSReleaseSocketMask`, `CSGetClientEventMask`, and `CSSetClientEventMask` to tell Card Services what events to forward.

The registration process provides the following services:

- It informs clients of cards already installed in a socket.
- It informs newly registered clients of any subsequent card events for a socket, such as card insertion, card removal, battery low, and so on.

The order in which clients are notified of card events is outlined in the Card Services section of *PCMCIA Standards*.

Programming Model

The PowerBook PC Card programming model supports the following features:

- *Parameter block programming interface.* Each function call to Card Services has a single parameter block argument. This is unlike the PCMCIA standard, which defines six arguments, one of which is a parameter block.
- *Interrupt/Event/Status change notification services.* Each driver or other client may request interrupt, event, or status change notification for a particular socket. The callback interface is described in the Card Services section of *PCMCIA Standards*.
- *Dynamic socket adapter registration.* Socket controller and Socket Services software can be added dynamically once Card Services is available. This allows Socket adapters to be engineered onto PDS™ (processor direct slot) cards or NuBus™ cards.
- *Single trap entry point.* All accesses to Apple PC Card support software are through the Card Services interface. Socket Services access is not available to clients, except for testing. A **glue routine** is provided for developers so that they can use the C function calling conventions.
- *Mostly synchronous calling environment.* The majority of calls to Card Services routines execute synchronously. For those executed asynchronously, the event callback mechanism notifies the caller when the operation is complete. There are no completion-routine requirements for calls to Card Services
- *Reentrancy.* Unlike Card Services, Socket Services is not reentrant code.

PowerBook Implementation of the PCMCIA Standard

The functional interface that exists in the PCMCIA standard is available to PC Card developers. However, elements of the standard that are not relevant to the PowerBook environment, or do not have direct analogies, are not included in the support architecture.

Although the PCMCIA standard is supposed to represent a platform-independent environment, it contains traces of the DOS architecture. The PowerBook implementation supports those DOS elements only where appropriate. For instance, function calls that allocate system resources for interrupt assignment in DOS do not have an analogous counterpart in the Mac OS, and normal Macintosh interrupt processing schemes are substituted.

CHAPTER 2

Client Software

A Card Services client is any third-party software that provides support for one or more types of PC Card. The software can take the form of a driver, extension, application, or other code. The client receives messages from Card Services indicating changes in the state of the PC Card(s) it controls, and makes access and resource requests to Card Services as needed for the card to perform its function.

The Mac OS Card Services implementation is compliant with version 2.01 of the PCMCIA standard. However, there are several divergences from the standard, mainly because of architectural differences between configurable voltage supplies to PC Cards, hardware windows into the card's address space, and interrupt decoding for state changes.

This chapter explains how to write client software for the Card Services API.

PCMCIA Services Model

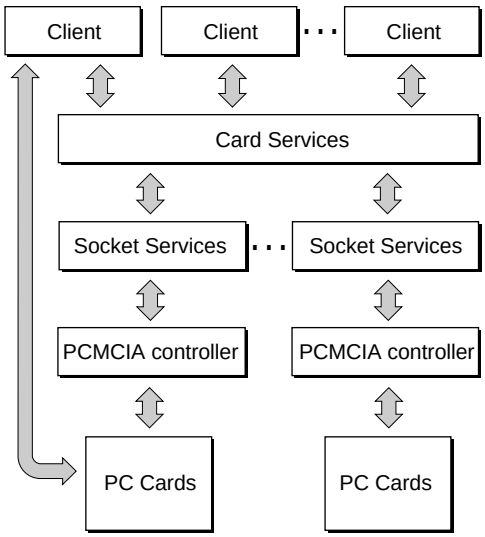
The PCMCIA standard provides a layered model to control card access and to isolate hardware specifics from client software. This makes it possible to write client software that can easily be ported to numerous PCMCIA platforms with few changes, if any.

The model consists of the following layers:

- **Clients** are responsible for performing functions associated with the particular kinds of cards with which they can deal.
- **Card Services** arbitrates access control for each card socket, manages card and PCMCIA resources, provides a client messaging system, and communicates with each of the installed Socket Services.
- **Socket Services** modules are software units that work directly with the PCMCIA hardware controller, converting generic commands into hardware-specific register accesses. They configure card power, windows, and pages, and dispatch interrupts to Card Services.
- The **PCMCIA controller** is the hardware interface to the PC Cards. It provides all the signals needed to communicate with the cards in the sockets it controls. In addition, it provides configurable voltages to power the cards, hardware windows into the card's address space, and interrupt decoding for state changes.
- Each **PC Card** contains a hardware interface to the PowerBook's PCMCIA bus as well as function-specific hardware. Cards may also have a **Card Information Structure (CIS).** This is a list of structures that describe the functions and capabilities of the card, and it is optional.

Figure 2-1 shows the interactions between the various components of the PCMCIA services model. Note that clients may communicate only with Card Services and, once they are allowed access, with the PC Card itself.

Figure 2-1 PCMCIA software/hardware model



Note: Each client may communicate directly with the PC Card(s) installed

Client Structure

A Card Services client can be any kind of software entity, including a driver, extension, application, and so forth. Every client has a client event handler. This is a routine, registered with Card Services, that processes notifications to the client of state changes to any of the cards it monitors. The client software registers its event handler with Card Services, which then calls this routine whenever it has an event for the client. The event handler's interface is

```
pascal UInt16 EventHandler(ClientCallbackPB* pb)
```

The client is passed

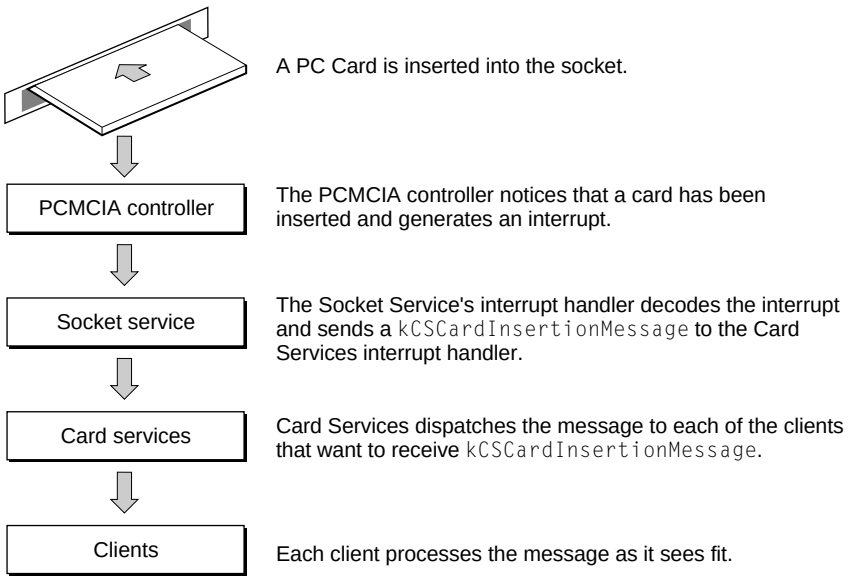
- a pointer to a parameter block containing information about the event
- message-specific data buffers
- a pointer to the client's data.

The client typically returns the result code `noErr`, although some messages may require that other result code values are returned instead.

Structure Overview

When there is a state change, such as a card insertion event, a process occurs, which starts from the physical insertion of the card and ends with the event handler. Figure 2-2 shows an example of this process.

Figure 2-2 Example of event progression



`kCSCardInsertionMessage` is one of several messages sent to a client's event handler. For more information about events, see "Event Processing" beginning on page 15.

The client's event handler is essentially an interrupt handler. Card state changes typically generate interrupts that are eventually forwarded to the client, as shown in Figure 2-2. Because of this, there are some precautions you should take when writing client software.

- Minimize the amount of time spent in the event handler, because interrupts could be restricted or disabled. This prevents the "jerky mouse" syndrome, as well as possible data loss in other software that does not get time. In many cases you can delay time-intensive processing until later.
- Use shared globals to communicate between the client event handler and the rest of the client software.
- When Virtual Memory (VM) is active, make sure the event handler's code and global data (not the entire client) is held in memory. This prevents paging at interrupt time.

Client Setup

Before a client can interact with Card Services, it must register its event handler so that Card Services can find the client. This is done by calling `CSRegisterClient`. The client passes a universal procedure pointer (UPP) to its event handler and a pointer to the client's data. Card Services adds the client to its queue, and synchronously initializes the client's state and returns. The parameter block passed to Card Services contains a `clientHandle` field. It is important to save this value in the client's globals because it is needed to identify the client in a number of calls to Card Services.

Note
The `CSRegisterClient` routine, and other routines mentioned in the following discussion, are described in Chapter 3, "Card Services Routines." ◆

The synchronous aspect of client registration represents a deviation from the PCMCIA standard. The standard requires `CSRegisterClient` to be asynchronous, with a `kCSRegistrationCompleteMessage` that notifies the client's event handler that registration is complete. Since this message may be sent at an arbitrary time, other messages may be sent to a client before registration is finished. The client is thus unable to process the message correctly at this time.

At this point, the client is registered with Card Services, but cannot receive any event messages. This prevents unexpected messages from being sent to the client before it has finished initialization. Once client initialization is complete, the client must call `CSVendorSpecific` with the `EnableSocketEvents` function code to notify Card Services that it is ready to receive messages. This is a Card Services call specific to the Mac OS.

The client should now be prepared to receive messages from Card Services. The remainder of the core client software consists of the client event handler.

Event Processing

When a hardware event results in a state change, or a software event is generated, a message is sent to each client's event handler. A pointer to a parameter block is passed to the event handler. This pointer contains the message (event type), message-specific buffer pointers, and a pointer to the client's global data. The client is free to process the message, and then return a result code to indicate what happened. Unless otherwise indicated, a result code of `noErr` is returned so that new types of messages will be handled correctly.

The event processing environment (not the events themselves) is binding-specific in conformance with the PCMCIA standard. Most events are dispatched at interrupt priority level 2, although some client-to-client events may be excluded. This means that event processing must adhere to the normal Mac OS interrupt handling restrictions described in *Inside Macintosh: Processes*, Chapter 1.

Any activity that cannot be handled in a normal interrupt handling environment (or that may take too much time to complete) should be delayed, using the techniques described in *Inside Macintosh: Processes*.

Some events (`kCSPMSuspendMessage`, `kCSPMResumeMessage`, and `kCSSSUpdatedMessage`) are not specific to any socket. These events are dispatched to clients based on the client's global event mask, and the client's `socketEvent` mask is irrelevant.

The next sections describe several types of events and what Card Services expects of them. *PCMCIA Standards* describes more types of events than those shown below. However, the ones described in this chapter merit particular attention because they are specific to the Mac OS, or because they are handled in a way that is slightly different from that described in *PCMCIA Standards*.

Card Insertion Message

A card insertion message (`kCSCardInsertionMessage`) is sent to all clients to notify them that a card has been inserted into a particular socket. By the time a client has received this message, power has been applied to the socket. However, the card is not ready for access at this point, so this call should be used only to indicate that the client may soon be able to talk to a card.

IMPORTANT

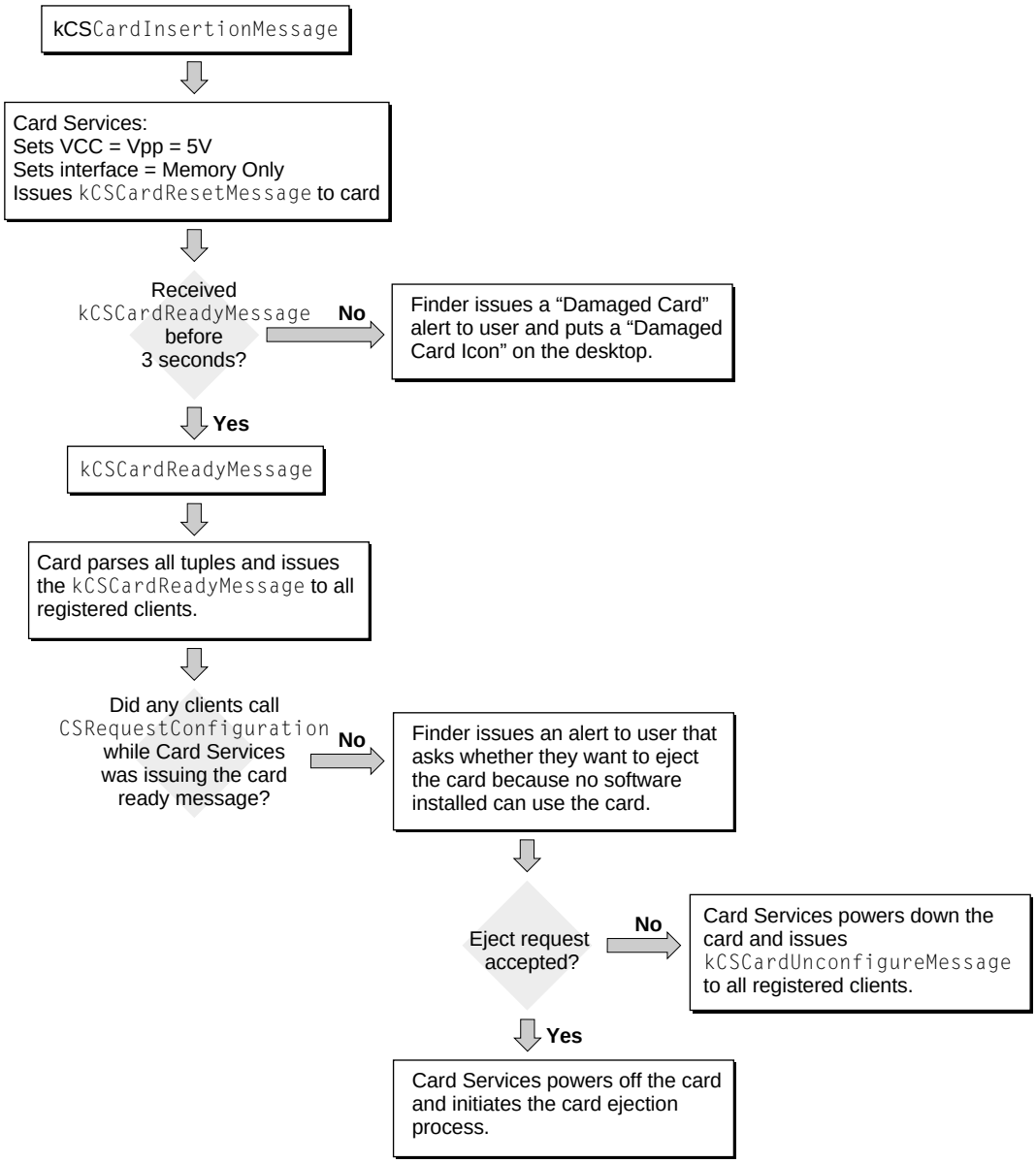
If you access the card at this time, even to read **tuples,** the access could cause a bus error. Wait until the client receives a `kCSCardReadyMessage` before trying to access the card. ▲

IMPORTANT

If a card fails to send a `kCSCardReadyMessage` within a prescribed period of time, clients should not post their own messages. Card Services or the Mac OS Finder Extension will deal with cards that appear to be damaged. ▲

Figure 2-3 illustrates the event processing that Card Services invokes when it receives `kCSCardInsertionMessage` from Socket Services.

Figure 2-3 Event processing from kCSCardInsertionMessage



Card Ready Message

When a client receives a card ready message (`kCSCardReadyMessage`), hardware on the card has determined that it is ready to be accessed. At this point, the client can start processing to determine whether or not it knows how to operate the card. It does this by calling `CSGetConfigurationInfo` to determine the card type, and to look for specific tuples that will indicate if this card is one with which the client can communicate.

If the client determines that it knows how to use this card, it calls `CSGetConfigurationInfo` to get the current card configuration, and then calls `CSRequestConfiguration` to request a card configuration. When the client does this, some or all of the card's resources are reserved for this client, and other clients will not be able to operate the card in a mode that interferes with the first client. You should check the result codes from each of these calls, since it is possible that another client has already reserved a configuration for the card.

IMPORTANT

In some situations a card may never become ready or it may take a long time to do so. If this happens, do nothing. The system software handles the timeouts from the point at which the card is inserted into the socket to the point at which it is ready for access. ▲

Card Removal Message

The card removal message (`kCSCardRemovalMessage`) indicates that a card has been removed. Card removal can occur at appropriate and inappropriate times. If the card has been legally ejected by calling `CSEjectCard`, then all clients should be safe and there should be no side affects.

If the card was removed manually, it may have been done when a client was in the process of accessing the card. In this case, a bus error is generated. Card Services does not provide any kind of bus error protection for clients doing card accesses, so individual clients must determine if they want to provide protection.

Ejection Request Message

Cards are ejected legally only after all clients have approved their ejection. This prevents a card from being removed at an inappropriate time. When a client calls `CSEjectCard`, each PC Card receives an ejection request message (`kCSEjectionRequestMessage`) giving it the opportunity to stop the card from being ejected. If the client allows the ejection, it should send a `noErr` result code. Otherwise it can return any other value to block the ejection.

Ejection Failed Message

The ejection failed message (`kCSEjectionFailedMessage`) is specific to the Mac OS and is not part of the PCMCIA standard. This message is sent to each client after ejection has been approved but the card cannot be successfully ejected. This may happen, for example, if the PCMCIA slot is physically blocked, or if there is a malfunction in the card eject mechanism.

Clients do not need to provide error notification to the user when this happens, since the system handles error notification.

Client Information Message

A client information message (`kCSClientInfoMessage`) is sent to the client's event handler whenever a `GetClientInfo` call is made using the client's `clientHandle`. A pointer to the `GetClientInfo` parameter block is passed to the client in the buffer field of the event handler's parameter block. The client can then determine what type of information is requested.

The upper byte of `GetClientInfoPB.attributes` contains a subfunction code that describes the kind of information being requested. The client sends a result code of `kCSUnsupportedModeErr` for any subfunctions it does not support.

Currently six subfunctions are supported. Except for the `csClientInfo` subfunction, they are all specific to the Mac OS. These subfunctions are described in the following sections.

kCSClientInfo Subfunction

This subfunction returns information about the client itself, such as the client's attributes and revision number, the lowest version of Card Services with which it is compliant, and C strings describing the client's function and manufacturer.

kCSCardName Subfunction

This subfunction returns an alternate name for a card other than that found in the card's tuples. This could be useful, for instance, if the name found on the card were something like "XY-D22-0354, REVISION 2.3," and you wanted to replace it with "FaxCo FaxModem," which provides a less cryptic human interface. The string is a zero-terminated C string.

kCSCardType Subfunction

This subfunction covers a range of functions. Serial card type covers serial devices such as UARTs, data and fax modems, pagers, and so on. The subfunction provides a means of returning a more descriptive string. For instance, it might replace the generic "Serial Port/Modem Card," with the more specific "Fax Modem Card." The string is a zero-terminated C string.

kCSHelpString Subfunction

This subfunction returns a descriptive help message which is used when the cursor moves over the card's icon. This message overrides the generic help message which is based on the card type. The string is a zero-terminated C string.

kCSCardIcon Subfunction

This subfunction returns a handle to an icon suite containing custom icons that display the card on the desktop. The client owns the icon suite, so users of the icon suite may not modify or dispose of it or its associated icons.

kCSActionProc Subfunction

When a client is called with this subfunction, it should perform a custom action in response to the user clicking or opening the card's icon. For instance, a pager card might open a custom pager application that can download messages stored in the pager and display them for the user. The client should send `noErr` result code if it performed the action. It should send any other value if it did not perform the action and wants the caller to handle it.

Function Interrupt Message

The function interrupt message (`kCSFunctionInterruptMessage`) is sent to the client's event handler in response to a function-specific interrupt from the card's hardware. The client then does any processing required by the interrupt, such as determining the interrupt source (for a multifunction card), reading or writing to a data register, and so on. The client then clears the source of the interrupt. This message is given high priority by Card Services to minimize interrupt service time as much as possible.

Power Management Suspend Message

Card Services issues a power management suspend message (`kCSPMSuspendMessage`) to indicate to registered clients that the PowerBook computer is going into sleep mode. This message is guaranteed to occur after all I/O transactions are complete. Clients should note the configuration of any cards and sockets they have configured because they must reestablish the configuration after the computer wakes from sleep mode.

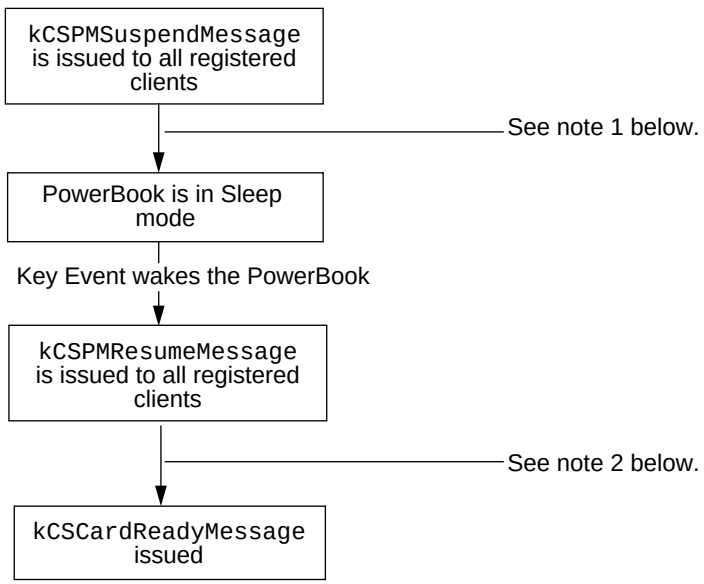
Power Management Resume Message

The power management resume message (`kCSPMResumeMessage`) is the first message a client receives when the computer is waking from sleep mode. This message happens before any I/O requests are generated. Immediately after Card Services has issued this message to all registered clients, it scans the socket configuration and supplies power to any socket that contains a PC Card but has no power. The process Card Services uses to supply power to these cards is identical to the process it uses when it receives a `kCSCardInsertionMessage` from Socket Services. Clients, therefore, should avoid accessing any cards until they have received a `kCSCardReadyMessage` for the socket.

Different PCMCIA controllers provide different information to Card Services when the PowerBook computer is waking from sleep mode. In some instances, Card Services may not be able to determine if PC Cards were switched in a socket during sleep mode. Clients should always provide some check to ensure that the card that appears in a socket is the same card the client was using before the computer went into sleep mode.

Figure 2-4 shows the processing sequence for `kCSPMSuspendMessage` and `kCSPMResumeMessage`.

Figure 2-4 kCSPMSuspendMessage and kCSPMResumeMessage processing



1. `kCSPMSuspendMessage` is issued to all registered clients after the last I/O access is completed prior to the computer entering sleep mode. Clients at this point should record any card configuration information they want to restore after the PowerBook wakes up. Clients should release all window resources, but keep all configurations. This means that they should not call `CSReleaseConfiguration`, because Card Services maintains the `clientHandle` configuration lock during sleep mode.
2. Card Services processes each socket with a card as if it were processing a `kCSCardInsertionMessage` event. Note that clients must not access the card until `kCSCardReadyMessage` has been received. `kCSPMResumeMessage` is merely an indication to clients that Card Services is attempting to awaken cards that were powered down during sleep mode. Card Services maintains the `clientHandle` configuration lock during sleep mode, but clients must reestablish the card/socket physical configurations by calling `CSModifyConfiguration` after they have received `kCSCardReadyMessage` for their socket.

Sample Client Code

This section provides code excerpts that illustrate some of the concepts discussed in the previous sections, including global variables, the client's initialization, removal, and event-handling codes.

Global Variables

The client needs shared global data to communicate state information between the event handler, which can run at interrupt time, and the rest of the client code that deals with the functional aspects of the card or cards under its control. The global variables can be set as follows:

```
typedef struct SharedGlobals SharedGlobals;

struct SharedGlobals
{
    .
    .
    .

  PCCardCSClientUPP  eventHandler;          // UPP for event handler
  unsigned long      clientHandle;          // returned by RegisterClient
  char               cardState[kMaxCards];  // state of each card
    .
    .
    .

};
```

The `eventHandler` field contains a UPP (universal procedure pointer) that points to the client's event handler. The `clientHandle` field contains the `clientHandle` returned by `CSRegisterClient`, which is used in a number of calls to Card Services. The `cardState` array contains the current state of each card in the system. It is a bitmap that contains the following bits:

```
#define kCardInserted          (1<<0)
#define kCardReady             (1<<1)
#define kCardIsMyType          (1<<2)
#define kCardIsWriteProtected  (1<<3)
#define kCardIsLocked          (1<<4)
#define kBatteryIsLow          (1<<5)
#define kBatteryIsDead          (1<<6)
```

With this minimal set of globals, you can now set up the client.

Client Initialization

The client initialization routine:

- allocates the shared globals
- registers the client's event handler with Card Services
- performs other initialization tasks specific to its needs

The following code is an example of an initialization routine:

```
void Initialize()
{
  SharedGlobals     *globals;
  RegisterClientPB  client;
  VendorSpecificPB  vsPB;

    .
    .
    .
// allocate space for the shared globals

  globals = (SharedGlobals *)NewPtrClear(sizeof(SharedGlobals));
  if (globals == nil) return;

  // create a Universal Proc Ptr that points to the event handler

  globals->eventHandler = NewPCCardCSClientProc(&MyEventHandler);

  // register the client with Card Services

  client.clientHandle = nil;
  client.clientEntry  = globals->clientEntry;
  client.attributes   = csExclusiveCardInsertEvents |
                        csExclusiveCardInsertEvents |
                        csIOClient;
  client.eventMask    = csReadyChangeEvent | csCardDetectChangeEvent;
  client.clientData   = (Ptr)globals;
  client.version      = 0;
  if (CSRegisterClient(&client) != noErr)
  {
    TearDownWorld(globals);
    return;
  }
  globals->clientHandle = client.clientHandle;

  // once everything's set up, make sure relevant pieces
  // are held in memory for VM
```

```
  if ((HoldMemory((void*)globals, sizeof(SharedGlobals)) != noErr) ||
      (HoldMemory((void*)&MyEventHandler, kEventHandlerSize) != noErr))
  {
    TearDownWorld(globals);
    return;
  }

  // now enable events from Card Services

    vsPB.clientHandle = globals->clientHandle;
    vsPB.vsCode       = vsEnableSocketEvents;

    if (CSVendorSpecific(&vsPB) != noErr)
 {
    TearDownWorld(globals);
    return;
 }

    .
    .
    .
}
```

Client Removal

The client removal routine is called when a client terminates in a normal manner, or when there is an error during client initialization. The following code is an example of a client removal routine:

```
void TearDownWorld(SharedGlobals *globals) {
  RegisterClientPB  client;

  // remove the event handler from Card Services' clutches

  client.clientHandle = globals->clientHandle;
  CSDeregisterClient(&client);

  // release the event handler's and globals' memory

  UnholdMemory((void*)globals, sizeof(SharedGlobals));
  UnholdMemory((void*)&MyEventHandler, kEventHandlerSize);

  // dispose of the globals

  DisposePtr((Ptr)globals);
}
```

Event Handler

The event handler routine is called by Card Services in response to a state change. Each message is stubbed out to show what it does, but a real client may require more work to be done for some messages, and may not need to do anything for others. The following code is an example of an event handler routine:

```
pascal short MyEventHandler(ClientCallbackPB *pb)
{
  SharedGlobals   *globals;
  unsigned short  socket;

  globals = (SharedGlobals *)pb->clientData;
  socket = pb->socket;

  switch (pb->function)
  {
    case kCSCardInsertionMessage:     // card has been inserted
      globals->cardState[socket] |= kCardInserted;
      break;

    case kCSCardRemovalMessage:        // card has been removed
      globals->cardState[socket] = 0;
      break;

    case kCSCardReadyMessage:          // card is ready to be accessed
      if (globals->cardState[socket] & kCardInserted)
        {
        globals->cardState[socket] |= kCardReady;
        if (CardIsMyType(socket, globals))
          globals->cardState[socket] |= kCardIsMyType;
      }
      break;

    case kCSEjectionRequestMessage:    // determine if it's OK to eject
                                       // my card
      if (globals->cardState[socket] & kCardIsMyType)
        return(CanCardBeEjected(socket, globals));
      break;

    case kCSEjectionFailedMessage:     // ejection failure (mechanism,
                                       // etc.)
      // depends on what the client wants to do
      break;

    case kCSClientInfoMessage:         // returns client information
      return(FillInClientInfo((GetClientInfoPB *)pb->buffer, globals));
```

```
    case kCSFunctionInterruptMessage:  // function interrupt from the
                                       // card
      HandleCardFunctionInterrupt(socket, globals);
      break;

    case kCSWriteProtectMessage:           // card is write-protected
      globals->cardState[socket] |= kCardIsWriteProtected;
      break;

    case kCSWriteEnabledMessage:         // card is write-enabled
      globals->cardState[socket] &= ~kCardIsWriteProtected;
      break;

    case kCSCardLockMessage:           // card is locked into the socket
      globals->cardState[socket] |= kCardIsLocked;
      break;

    case kCSCardUnlockMessage:         // card's socket is now unlocked
      globals->cardState[socket] &= ~kCardIsLocked;
      break;

    case kCSBatteryDeadMessage:        // card's battery is dead
      globals->cardState[socket] |= kBatteryIsDead;
      break;

    case kCSBatteryLowMessage:         // card's battery is getting low
      globals->cardState[socket] |= kBatteryIsLow;
      break;

    case kCSCardResetMessage:
    case kCSInsertionRequestMessage:
    case kCSInsertionCompleteMessage:
    case kCSPMResumeMessage:
    case kCSPMSuspendMessage:
    case kCSExclusiveRequestMessage:
    case kCSExclusiveCompleteMessage:
    case kCSResetPhysicalMessage:
    case kCSResetRequestMessage:
    case kCSResetCompleteMessage:
    case kCSSSUpdatedMessage:
          break;
  }
  return(noErr);
}

#define  kClientVersion       0x0100    // driver version, in BCD
#define  kCardServicesLevel   0x0201    // Card Services release
                                        // compliance, in BCD
#define  kRevisionDate      ((1994-1980)<<9) | (8<<5) | (25<<0)  // y/m/d
```

Returning Client Information

When a call is made to `CSGetClientInfo`, Card Services calls the client event handler associated with the `clientHandle` passed in the parameter block. `kCSClientInfoMessage` is passed to the client event handler, and `ClientCallbackPB.buffer` contains a pointer to a `GetClientInfoPB`.

Driver Location Icon

Mass storage device drivers, such as those for hard disk and floppy disk drives, return a pointer to a structure that describes the location of their device. This process is handled by a control call with `csCode` set to 22. The structure is:

```
typedef struct DriverLocationIcon DriverLocationIcon;
struct DriverLocationIcon
{
  char        locationIcon[256];
  Str255      locationString;
};
```

For most mass storage devices, the driver knows where its device resides physically. This means it can easily return an appropriate icon and location string. For PCMCIA-based mass storage devices, the driver does not intrinsically know where its card is plugged in, so it cannot provide a location icon.

The PowerBook implementation of Card Services provides a means for clients (typically drivers) to acquire both the card's location icon and a string describing the card's current location. Clients can call `CSVendorSpecific` with `pb.vsCode` set to either `vsGetCardLocationIcon` or `vsGetCardLocationText`, and Card Services will return the appropriate information.

▲ **WARNING**

The location icon and string can be localized, so that they can be loaded from a disk. In the same way, Socket Services modules can be loaded at any time during the boot process. Clients, therefore, should not assume that the location information is valid before the PowerBook computer has finished the start-up process. ▲

Because location information is not returned in the exact format that the driver control call expects, the driver must pack the information into the format described above. For example, in response to control call 22, the driver can make this call:

```
OSErr DriverControl(CntrlParam *pb, DCtlHandle dce)
{
    .
    .
    .
  switch (pb->csCode)
```

```
  {
    .
    .
    .
   case 22:
      socket = GetDriveSocket(globals, pb->ioVRefNum);
      GetDriverLocation(socket, &globals->locationIcon);
      *(DriverLocationIcon *)&pb->csParam[0] = &globals->locationIcon;
      return(noErr);
    .
    .
    .
     }
    .
    .
    .
}
```

`GetDriverLocationIcon` makes the following calls to `CSVendorSpecific` to get the icon and string:

```
void GetDriverLocationIcon(unsigned short socket,
                           DriverLocationIcon *theIcon)
{
  VendorSpecificPB  pb;
  Handle            theIconSuite, theIconData;
  Str255            locationString;

  pb.vsCode    = vsGetCardLocationIcon;
  pb.socket    = socket;
  pb.dataLen   = 0;
  pb.vsDataPtr = (unsigned char *) &theIconSuite;
  if ((CSVendorSpecific(&pb) == noErr) && (theIconSuite != nil) &&
      (GetIconFromSuite(&theIconData, theIconSuite, 'ICN#') != noErr))
       BlockMove(*theIconData, theIcon->locationIcon, 256);

  pb.vsCode    = vsGetCardLocationText;
  pb.socket    = socket;
  pb.dataLen   = 256;
  pb.vsDataPtr = (Ptr) &theIcon->locationString;
  CSVendorSpecific(&pb);
  c2pstr(theIcon->locationString);
}
```

Note

The `vsGetCardLocationText` call follows the Card Services convention for returning C strings, which are terminated with a null character. It is necessary to call `c2pstr()` on the returned string to convert it to a Pascal string, which contains a leading length byte. ◆

Sample Client Pseudocode

The first task a client undertakes is to recognize Card Services and determine its compatibility with the Card Services version. If a client can interact with the available Card Services, it registers with Card Services and provides the address of its event handler, as shown below. The handler must be locked down while the client is registered.

```
main()
{
   RegisterClientPB client;

   If ((Card Services Exists) && (Card Services Revision is appropriate))
   {
      client.clientHandle = nil;
      client.clientEntry  = &ClientEventHandler;
      .
      .
      .
      CSRegisterClient(&client);

      gClientHandle = client.clientHandle;

      CSVendorSpecific(vsCode == vsEnableSocketEvents);
   }
}
```

The event handler consists of the following big switch statement:

```
ClientEventHandler(ClientCallbackPB pb)
{
   switch(pb->event)
   {
      case kCSCardInsertionMessage:
         DoCardInsertion();
         break;

      case kCSCardReadyMessage:
         DoCardReady(pb->socket);
         break;

      case kCSCardRemovalMessage:
         DoCardRemoval();
         break;

      case kCSEjectionRequestMessage:
         DoEjectionRequest();
         break;
```

```
        case kCSPMSuspendMessage:
            DoPMSuspend();
            break;

        .
        .
        .
    }
}
```

The normal sequence of events delivered to a client upon the insertion of a PC Card into a socket is `kCSCardInsertionMessage` -> `kCSCardReadyMessage` -> . . .

Clients should not call any Card Services function that requires a card access until `kCSCardReadyMessage` is delivered. Upon receiving `kCSCardReadyMessage`, clients are free to parse the card attribute space (for example, using `CSGetFirstTuple`, `CSGetNextTuple`, and `CSGetTupleData`) or ask Card Services for condensed information about the card (for example, using `CSVendorSpecific(vsCode == vsGetCardInfo)`).

When a client recognizes an inserted card it typically performs the following tasks:

```
DoCardReady(socket)
{
    GetModRequestConfigInfoPB  getModReqConfigPB;
    ReqModRelWindowPB          reqModRelWindowPB;

    if (CSGetConfigurationInfo(&getModReqConfigPB) != noErr)
        return(err);

    if (getModReqConfigPB.firstDevType != myDeviceType)
        return(err);

    getModReqConfigPB.clientHandle = gClientHandle;
    getModReqConfigPB.socket = socket;

    // we want leave the card in Memory Only mode (when we get
kCSCardReadyMessage
    //  for the first time the interface and card are in Memory Only mode)

    if (result = CSRequestConfiguration(&getModReqConfigPB))
        return(result);

    // map the card into a window

    reqModRelWindowPB.clientHandle = gClientHandle;
    reqModRelWindowPB.windowHandle = nil;
    reqModRelWindowPB.socket = socket;
```

```
    reqModRelWindowPB.attributes =...;
    reqModRelWindowPB.base = 0;
    reqModRelWindowPB.size = 0;

    if (result = CSRequestWindow(&reqModRelWindowPB))
        return(result);

    return;
}
```

CHAPTER 3

Card Services Routines

The PC Card Manager helps client software recognize, configure, and view PC Cards that are inserted into PC Card sockets on PowerBook computers. The PC Card Manager is composed of two sets of system software:

- Card Services is used by all PC Card client software. It is a new part of the Mac OS and allows software to use PC Cards.
- Socket Services is used primarily by developers of new PC Card hardware.

This chapter describes only the Card Services routines as they apply to PC Cards used in PowerBook computers. The chapter provides the following information for each routine: a general description, software code, and result codes. When applicable, it supplies supplementary information about the routine and also indicates areas where the routine differs from the PCMCIA standard.

For detailed information on both Card Services and Socket Services, refer to *PCMCIA Standards*, published by the Personal Computer Memory Card International Association.

Client Information

The routines described in this section let you access Card Services clients and get information about those clients.

CSGetFirstClient

This routine returns a client handle for the first client in the Card Services global client first-in-first-out (FIFO) queue.

```
pascal OSErr CSGetFirstClient(GetClientPB *pb)

typedef struct GetClientPB GetClientPB;
struct GetClientPB
{
   UInt32  clientHandle;  // <-  clientHandle for this client
   UInt16  socket;        //  -> logical socket number
   UInt16  attributes;    //  -> bitmap of attributes
};

// 'attributes' field values

enum
{
   kCSClientsForAllSockets = 0x0000,
   kCSClientsThisSocketOnly= 0x0001
};
```

SUPPLEMENTARY INFORMATION

The client handle returned by this routine is used with `CSGetClientInfo`. If the caller specifies `kCSClientsThisSocketOnly` and passes in a valid socket number, Card Services returns the first client whose event mask for the given socket is not `NULL`.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoMoreItemsError`	No clients registered

CSGetNextClient

This routine returns a client handle for the next client in the Card Services global first-in-first-out (FIFO) queue.

```
pascal OSErr CSGetNextClient(GetClientPB *pb)

typedef struct GetClientPB GetClientPB;
struct GetClientPB
{
   UInt32  clientHandle;  // <-> clientHandle for this client
   UInt16  socket;        //  -> logical socket number
   UInt16  attributes;    //  -> bitmap of attributes
};
```

For the field values of `attributes`, see “CSGetFirstClient” on page 34.

SUPPLEMENTARY INFORMATION

The next client handle is used with `CSGetClientInfo`. If the caller specifies `kCSClientsThisSocketOnly` and passes in a valid socket number, Card Services returns the next client whose event mask for the given socket is not `NULL`.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoMoreItemsError`	No clients registered
`kCSBadHandleErr`	Invalid client handle

CSGetClientInfo

This routine returns information that describes a client, and it may be used by browsing utilities. The information returned includes items such as revision date, length of the client name, logical socket number, and so forth.

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

```
typedef struct ClientInfoParam ClientInfoParam;
struct ClientInfoParam
{
          UInt32 clientHandle;    //  -> clientHandle returned by
                                      RegisterClient
          UInt16 attributes;      // <-> subfunction + bitmapped client
                                      attributes
          UInt16  revision;       // <-  BCD value of client's revision
          UInt16  kCSLevel;       // <-  BCD value of CS release
          UInt16  revDate;        // <-  revision date: y[15-9], m[8-5], d[4-0]
          SInt16  nameLen;        // <-> in: max length of client name string,
                                  //     out: actual length
          SInt16  vStringLen;     // <-> in: max length of vendor string,
                                  //     out: actual length
          UInt8   *nameString ;;  // <-  pointer to client name string
                                  //     (zero-terminated)
          UInt8   *vendorString;  // <-  pointer to vendor string
                                  //     (zero-terminated)
      };
      // upper byte of attributes is
                          // kCSCardNameSubfunction,
      {                   // kCSCardTypeSubfunction,
                          // kCSHelpStringSubfunction

typedef struct AlternateTextStringParam AlternateTextStringParam;
struct AlternateTextStringParam
{
          UInt32 clientHandle; //  -> clientHandle returned by
                                     RegisterClient
          UInt16 attributes;   // <-> subfunction + bitmapped client
                                     attributes
          UInt16  socket;      //  -> logical socket number
          UInt16  reserved;//  -> zero
          SInt16  length;  // <-> in: max length of string,
                           //     out: actual length
          UInt8   *text;   // <-  pointer to string (zero-terminated)
      };
   // upper byte of attributes is
                          //kCSCardIconSubfunction
typedef struct AlternateCardIconParam AlternateCardIconParam;
```

```
      struct AlternateCardIconParam
      {
         UInt32 clientHandle; //  -> clientHandle returned by
                                 RegisterClient
         UInt16 attributes;   // <-> subfunction + bitmapped client
                                 attributes
         UInt16 socket;       //  -> logical socket number
         UInt16 reserved;     //  -> zero
         Handle  iconSuite;   // <-  handle to suite containing all icons
      }

   //upper byte of attributes is
                          //kCSActionProcSubfunction
typedef struct CustomActionProcParam CustomActionProcParam
struct              // upper struct CustomActionProcParam
{
         UInt32 clientHandle;    //-> clientHandle returned by
                                   RegisterClient
         UInt16 attributes;      // <-> subfunction + bitmapped client
                                        attributes
         UInt16  socket;         //  -> logical socket number
         };
typedef struct GetClientInfoPB GetClientInforPB;
struct GetClientInfoPB {
union {
   ClientInfoParam          clientInfo;
   AlternateTextStringParam alternateTextString;
   AlternateCardIconParam   alternateIcon;
   CustomActionProcParam    customActionProc;
}                         u;
};

// 'attributes' field values

enum {
   kCSMemoryClient              = 0x0001,
   kCSIOClient                  = 0x0004,
   kCSClientTypeMask            = 0x0007,
   kCSShareableCardInsertEvents  = 0x0008,
   kCSExclusiveCardInsertEvents  = 0x0010,

   kCSInfoSubfunctionMask       = 0xFF00,
   kCSClientInfoSubfunction     = 0x0000,
   kCSCardNameSubfunction       = 0x8000,
   kCSCardTypeSubfunction       = 0x8100,
   kCSHelpStringSubfunction     = 0x8200,
   kCSCardIconSubfunction       = 0x8300,
   kCSActionProcSubfunctionon    = 0x8400
};
```

SUPPLEMENTARY INFORMATION

The `CSGetClientInfo` routine is used by clients to extract information about the client whose client handle is passed in. Note that in this case the caller does not pass in its own client handle. Card Services passes `kCSClientInfoMessage` to the client pointed to by the client handle. The caller of `CSGetClientInfo` passes the requested information subfunction in the upper byte of the `attributes` field. Called clients should respond to `kCSClientInfoMessage` by filling out the data requested. When a client receives `kCSClientInfoMessage` that requires it to perform a custom action, it should be aware that it is being called from the Finder or a similar process environment.

Each client that is called with `kCSClientInfoMessage` is passed one of the items from `ClientCallbackPB`. The buffer field of `ClientCallbackPB` contains a pointer to `GetClientInfoPB`:

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

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

Callers of `CSGetClientInfo` should use `GetFirstClient` and `GetNextClient` to iterate through all the registered clients. Card Services passes the client handle to the caller of either routine.

RESULT CODES

`noErr`	No error
`kCSBadHandleErr`	Invalid client handle

Configuration Routines

The routines described in this section help you to configure the PC Cards and the 68-pin sockets into which they are plugged.

CSGetConfigurationInfo

This routine returns information about the specified socket and PC Card configuration. The information, as shown below, includes such things as status register setting, device ID, and manufacturer's ID.

```
pascal OSErr CSGetConfigurationInfo(GetModRequestConfigInfoPB *pb)

typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  socket;        //  -> logical socket number
   UInt16  attributes;    // <-  bitmap of configuration attributes
   UInt8   vcc;           // <-  Vcc setting
   UInt8   vpp1;          // <-  Vpp1 setting
   UInt8   vpp2;          // <-  Vpp2 setting
   UInt8   intType;       // <-  interface type (memory or memory+I/O)
   UInt32  configBase;    // <-  card base address of config registers
   UInt8   status;        // <-  card status register setting, if present
   UInt8   pin;           // <-  card pin register setting, if present
   UInt8   copy;          // <-  card socket/copy reg setting, if present
   UInt8   configIndex;   // <-  card option register setting, if present
   UInt8   present;       // <-  bitmap of which config regs are present
   UInt8   firstDevType;  // <-  from DeviceID tuple
   UInt8   funcCode;      // <-  from FuncID tuple
   UInt8   sysInitMask;   // <-  from FuncID tuple
   UInt16  manufCode;     // <-  from ManufacturerID tuple
   UInt16  manufInfo;     // <-  from ManufacturerID tuple
   UInt8   cardValues;    // <-  valid card register values
   UInt8   padding[1];
};

// 'attributes' field values

enum
{
   kCSExclusivelyUsed     = 0x0001,
   kCSEnableIREQs         = 0x0002,
   kCSVccChangeValid      = 0x0004,
   kCSVpp1ChangeValid     = 0x0008,
   kCSVpp2ChangeValid     = 0x0010,
   kCSValidClient         = 0x0020,
   // request that power be applied to socket during sleep
   kCSSleepPower          = 0x0040,
```

```
    kCSLockSocket          = 0x0080,
    kCSTurnOnInUse         = 0x0100
};

// 'intType' field values
enum
{
    kCSMemoryInterface         = 0x01,
    kCSMemory_And_IO_Interface = 0x02
};

// 'present' field values

enum
{
    kCSOptionRegisterPresent         = 0x01,
    kCSStatusRegisterPresent         = 0x02,
    kCSPinReplacementRegisterPresent = 0x04,
    kCSCopyRegisterPresent           = 0x08
};

// 'cardValues' field values

enum
{
    kCSOptionValueValid       = 0x01,
    kCSStatusValueValid       = 0x02,
    kCSPinReplacementValueValid= 0x04,
    kCSCopyValueValid         = 0x08
};
```

SUPPLEMENTARY INFORMATION

The `CSGetConfigurationInfo` routine is generally called after a client has parsed a tuple stream, identified an inserted card as its card, and then wants to initialize a `GetModRequestConfigInfoPB` parameter block. For a typical sequence of events, see "Card Ready Message" beginning on page 18. For information about tuples, refer to "Tuple Information" on page 49, and to the Glossary at the end of this note.

RESULT CODES

`noErr`	No error
`kCSBadHandleErr`	Invalid client handle

CSRequestConfiguration

A client uses this routine to configure the PC Card and the socket. The routine must be used by clients that require an I/O interface to their PC Card.

```
pascal OSErr CSRequestConfiguration(GetModRequestConfigInfoPB *pb)

typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
   UInt32 clientHandle;//  -> clientHandle returned by RegisterClient
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> bitmap of configuration attributes
   UInt8  vcc;         //  -> Vcc setting
   UInt8  vpp1;        //  -> Vpp1 setting
   UInt8  vpp2;        //  -> Vpp2 setting
   UInt8  intType;     //  -> interface type (memory or memory+I/O)
   UInt32 configBase;  //  -> card base address of configuration registers
   UInt8  status;      //  -> card status register setting, if present
   UInt8  pin;         //  -> card pin register setting, if present
   UInt8  copy;        //  -> card socket/copy register setting, if present
   UInt8  configIndex; //  -> card option register setting, if present
   UInt8  present;     //  -> bitmap of which confign registers are present
   UInt8  firstDevType;// <-  from DeviceID tuple
   UInt8  funcCode;    // <-  from FuncID tuple
   UInt8  sysInitMask; // <-  from FuncID tuple
   UInt16 manufCode;   // <-  from ManufacturerID tuple
   UInt16 manufInfo;   // <-  from ManufacturerID tuple
   UInt8  cardValues;  // <-  valid card register values
   UInt8  padding[1];  //
};
```

For the field values of `attributes`, `intType`, `present`, and `cardValues`, see "CSGetConfigurationInfo" beginning on page 39.

SUPPLEMENTARY INFORMATION

A client calls `CSRequestConfiguration` after it has parsed a PC Card that is inserted and ready, and has recognized the card as being usable.

Card Services uses the client handle to maintain a lock on the configuration until the same client calls `CSReleaseConfiguration`. Once a socket and card are configured, no other client may alter their configuration.

Configuring a socket and card requires three operations:

- establishing Vcc and Vpp for the socket
- establishing the socket interface definition (Memory Only or I/O and Memory)
- writing to the configuration registers on the card

When Card Services receives `KCSCardInsertionMessage` and subsequently `kCSCardReadyMessage` for a socket, it configures the socket by setting Vcc, Vpp1, and Vpp2 to 5 volts, configuring the interface to be Memory Only, and issuing a `RESET` to the card. Card Services then parses the CIS of the card. Once Card Services has finished parsing the CIS, it issues `kCSCardReadyMessage` to all registered clients. It has already delivered `kCSCardInsertionMessage` to the same clients. Even if a client parses and recognizes a card and intends to use the card without altering the configuration, it should call `CSRequestConfiguration` to establish it as the configuring client.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle
`kCSConfigurationLockedErr`	Another client has already locked a configuration
`kCSNoCardErr`	No card in the specified socket
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSBadBaseErr`	Invalid base entered

CSModifyConfiguration

This routine allows a socket and PC Card configuration to be modified without using the `CSReleaseConfiguration` routine followed by the `CSRequestConfiguration` routine. The routine can only modify a configuration originally requested through `CSRequestConfiguration`.

```
pascal OSErr CSModifyConfiguration(GetModRequestConfigInfoPB *pb)

typedef struct GetModRequestConfigInfoPB GetModRequestConfigInfoPB;
struct GetModRequestConfigInfoPB
{
   UInt32 clientHandle;   //  -> clientHandle returned by RegisterClient
   UInt16 socket;         //  -> logical socket number
   UInt16 attributes;     //  -> bitmap of configuration attributes
   UInt8  vcc;            //  -> Vcc setting
   UInt8  vpp1;           //  -> Vpp1 setting
   UInt8  vpp2;           //  -> Vpp2 setting
   UInt8  intType;        //  -> interface type (memory or memory+I/O)
   UInt32 configBase;     //  -> card base address of config registers
   UInt8  status;         //  -> card status register setting, if present
   UInt8  pin;            //  -> card pin register setting, if present
   UInt8  copy;           //  -> card socket/copy regr setting, if present
   UInt8  configIndex;    //  -> card option register setting, if present
```

```
    UInt8  present;        //  -> bitmap of which config regis are present
    UInt8  firstDevType;   // <-  from DeviceID tuple
    UInt8  funcCode;       // <-  from FuncID tuple
    UInt8  sysInitMask;    // <-  from FuncID tuple
    UInt16 manufCode;      // <-  from ManufacturerID tuple
    UInt16 manufInfo;      // <-  from ManufacturerID tuple
    UInt8  cardValues;     // <-  valid card register values
    UInt8  padding[1];     //
};
```

For `'attributes'`, `'intType'`, `'present'`, and `'cardValues'` field values see "CSGetConfigurationInfo" beginning on page 39.

SUPPLEMENTARY INFORMATION

The `CSModifyConfiguration` routine is used by clients to alter any of the three configuration elements of a socket or card: the power supply (Vcc and Vpp) for the socket, the socket interface definition (Memory Only or I/O and Memory), the configuration registers on the card. Only a client that has previously succeeded in calling `CSRequestConfiguration` may call `CSModifyConfiguration`.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle
`kCSConfigurationLockedErr`	Another client has already locked a configuration
`kCSNoCardErr`	No card in the specified socket
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSBadBaseErr`	Invalid base entered

CSReleaseConfiguration

This routine releases the PC Card and socket from the I/O interface, and returns them to a memory-only interface with configuration 0. If no clients have indicated that they are using the socket, Card Services may remove power from the socket.

```
pascal OSErr CSReleaseConfiguration(ReleaseConfigurationPB *pb)

typedef struct ReleaseConfigurationPB ReleaseConfigurationPB;
struct ReleaseConfigurationPB
{
    UInt32  clientHandle;
    UInt16  socket;
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle
`kCSConfigurationLockedErr`	Another client has already locked a configuration
`kCSNoCardErr`	No card in the specified socket

CSAccessConfigurationRegister

This routine allows a client to modify a single configuration register. It can do this by adding `AccessConfigurationRegisterPC.offset` to the configuration base address. However, clients do not generally use this routine.

```
pascal OSErr CSAccessConfigurationRegister(AccessConfigurationRegisterPB *pb)
```

```
typedef struct AccessConfigurationRegisterPB AccessConfigurationRegisterPB;

struct AccessConfigurationRegisterPB
{
   UInt16 socket;            //  -> global socket number
   UInt8  action;            //  -> read/write
   UInt8  offset;            //  -> offset from config register base
   UInt8  value;             // <-> value to read/write
   UInt8  padding[1];
};

// 'action' field values
enum {
   CS_ReadConfigRegister  = 0x00,
   CS_WriteConfigRegister = 0x01
};
```

SUPPLEMENTARY INFORMATION

If the client uses this routine to modify a register, and adds `AccessConfigurationRegisterPB.offset` to the configuration base address, then one of two things happens:

- If `action` is set to `CS_ReadConfigRegister`, the configuration register value is returned in `AccessConfigurationRegisterPB.value`.
- If `action` is set to `CS_WriteConfigRegister`, the configuration register is written with `AccessConfigurationRegisterPB.value`.

When clients want to set configuration registers, they usually call `CSRequestConfiguration` or `CSModifyConfiguration` and set the appropriate registers at that time.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified

Masking Routines

The routines described in this section get and set masks for client events and sockets. Card Services provide notification about events based on the contents of this field. Each mask is a 16-bit field, with bit 0 being the lowest-order bit. In the case of event masks, the 8 lower-order bits specify events noted by Socket Services, and the 8 higher-order bits specify events generated by Card Services. The socket masks establish or clear an event mask for a given socket number.

CSGetClientEventMask

Clients use this routine to obtain their current event mask.

```
pascal OSErr CSGetClientEventMask(GetSetClientEventMaskPB *pb)

typedef struct GetSetClientEventMaskPB GetSetClientEventMaskPB;
struct GetSetClientEventMaskPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  attributes;    //  -> bitmap of attributes
   UInt16  eventMask;     // <-  bitmap of events to be passed to client for
                          //     this socket
   UInt16  socket;        //  -> logical socket number
};

// 'attributes' field values

enum
{
   kCSEventMaskThisSocketOnly = 0x0001
};

// 'eventMask' field values

enum
{
   kCSWriteProtectEvent       = 0x0001,
   kCSCardLockChangeEvent     = 0x0002,
   kCSEjectRequestEvent       = 0x0004,
```

```
    kCSInsertRequestEvent      = 0x0008,
    kCSBatteryDeadEvent        = 0x0010,
    kCSBatteryLowEvent         = 0x0020,
    kCSReadyChangeEvent        = 0x0040,
    kCSCardDetectChangeEvent   = 0x0080,
    kCSPMChangeEvent           = 0x0100,
    kCSResetEvent              = 0x0200,
    kCSSSUpdateEvent           = 0x0400,
    kCSFunctionInterrupt       = 0x0800,
    kCSAllEvents               = 0xFFFF
};
```

SUPPLEMENTARY INFORMATION

If `GetSetClientEventMaskPB.attributes` has `kCSEventMaskThisSocketOnly` (bit 0) reset, `CSGetClientEventMask` returns the client's global event mask. If `GetSetClientEventMaskPB.attributes` has `kCSEventMaskThisSocketOnly` set, then the event mask for the given socket number is returned. If the client has not registered for the socket, `kCSBadSocketErr` is returned.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle

CSSetClientEventMask

Clients use this routine to establish their event masks.

```
pascal OSErr CSSetClientEventMask(GetSetClientEventMaskPB *pb)
```

```
typedef struct GetSetClientEventMaskPB GetSetClientEventMaskPB;
struct GetSetClientEventMaskPB
{
    UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
    UInt16  attributes;    //  -> bitmap of attributes
    UInt16  eventMask;     //  -> bitmap of events to be passed to client
                           //     for this socket
    UInt16  socket;        //  -> logical socket number
};
```

For the field values of `eventMask`, see "CSGetClientEventMask" on page 45.

SUPPLEMENTARY INFORMATION

If `GetSetClientEventMaskPB.attributes` (bit 0) is reset, `CSSetClientEventMask` (the client's global event mask) is changed. If `GetSetClientEventMaskPB.attributes` has `kCSEventMaskThisSocketOnly` set, then the event mask for the given socket number is changed. If the client has not registered for the socket, `kCSBadSocketErr` is returned.

After processing `kCSCardReadyMessage` and determining that the card is not usable, clients should clear their global event masks so that message processing with the system is streamlined.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle

CSRequestSocketMask

This routine requests that the client be notified of status changes for the given socket. If the client has events enabled in the global event mask, it may be notified more than once for each status change for the socket.

```
pascal OSErr CSRequestSocketMask(ReqRelSocketMaskPB *pb)

typedef struct ReqRelSocketMaskPB ReqRelSocketMaskPB;
struct ReqRelSocketMaskPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  socket;        //  -> logical socket
   UInt16  eventMask;  //  -> bitmap of events to be passed to client for
                       //     this socket
};
```

For the field values of `eventMask`, see "CSGetClientEventMask" on page 45.

SUPPLEMENTARY INFORMATION

`CSRequestSocketMask` must be used before `CSSetClientEventMask` or `CSGetClientEventMask`. Otherwise, these two routines may not execute successfully. If the client has not registered for the socket, `kCSBadSocketErr` is returned.

RESULT CODES

noErr	No error
kCSBadSocketErr	Invalid socket specified
kCSBadHandleErr	Invalid client handle

CSReleaseSocketMask

The client uses this routine to clear the event mask and to request that it no longer be notified of status changes to the socket. However, if the client has events enabled in a global event mask, it will still be notified of status changes in that mask. This is the recommended way for clients to clear socket events when they recognize that they are not interested in using a particular PC Card.

```
pascal OSErr CSReleaseSocketMask(ReqRelSocketMaskPB *pb)
```

```
typedef struct ReqRelSocketMaskPB ReqRelSocketMaskPB;
struct ReqRelSocketMaskPB
{
   UInt32  clientHandle;//  -> clientHandle returned by RegisterClient
   UInt16  socket;     //  -> logical socket
   UInt16  eventMask;  //  -> bitmap of events to be passed to client for
                       //     this socket
};
```

For the field values of `eventMask`, see "CSGetClientEventMask" on page 45.

SUPPLEMENTARY INFORMATION

`CSReleaseSocketMask` is used to clear the event mask for the given socket. If the client has not registered for the socket, `kCSBadSocketErr` is returned.

RESULT CODES

noErr	No error
kCSBadSocketErr	Invalid socket specified
kCSBadHandleErr	Invalid client handle

Tuple Information

The routines described in this section allow you to access tuples. A tuple is a chain or linked list of data blocks. It can be of various lengths. Tuples contain information about the PC Card, such as access speed, function ID, manufacturer's ID, organization, and so forth.

CSGetFirstTuple

This routine returns the first tuple of the type specified in the CIS for the specific socket. If there are no tuples, the status argument returns `kCSNoMoreItemsErr`. The first tuple identifies the local socket containing the specified PC Card. It also contains the information needed to link to the next tuple, as well as fields that are used internally by Card Services. Tuple format is described in more detail in "CSGetTupleData" on page 52.

```
pascal OSErr CSGetFirstTuple(GetTuplePB *pb)

typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> bitmap of attributes
   UInt8  desiredTuple;//  -> desired tuple code value, or $FF for all
   UInt8  tupleOffset; //  -> offset into tuple from link byte
   UInt16 flags;       // <-> internal use only!
   UInt32 linkOffset   // <-> internal use only!
   UInt32 cisOffset;   // <-> internal use only!

   union
   {
      struct
      {
         UInt8  tupleCode;   // <-  tuple code found
         UInt8  tupleLink;   // <-  link value for tuple found
      }  TuplePB;

      struct
      {
         UInt16     tupleDataMax;//  -> maximum size of tuple data area
         UInt16     tupleDataLen;// <-  number of bytes in tuple body
         TupleBody  tupleData;  // <-  tuple data
```

```
      }  TupleDataPB;
   } u;
};

// 'attributes' field values
enum
{
   kCSReturnLinkTuples = 0x0001
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSInUseErr`	Card is configured and being used by another client
`kCSReadFailureErr`	Card cannot be read
`kCSBadCISErr`	Card Services has encountered a bad CIS structure
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSNoMoreItemsErr`	There are no more tuples to process

CSGetNextTuple

This routine returns the next tuple of the type specified in the CIS for the specific socket. If there are no tuples, the status argument returns `kCSNoMoreItemsErr`. The next tuple contains information about the link value for the tuple. Certain fields in the tuple are reserved for internal use by Card Services. These fields are updated by Card Services, and must have the same values as the tuple returned just previously. Tuple format is described in more detail in “CSGetTupleData” on page 52.

```
pascal OSErr CSGetNextTuple(GetTuplePB *pb)

typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> bitmap of attributes
   UInt8  desiredTuple;//  -> desired tuple code value, or $FF for all
   UInt8  tupleOffset; //  -> offset into tuple from link byte
```

```
    UInt16 flags;       // <-> internal use only!
    UInt32 linkOffset;  // <-> internal use only!
    UInt32 cisOffset;   // <-> internal use only!

    union
    {
        struct
        {
            UInt8  tupleCode;   // <-  tuple code found
            UInt8  tupleLink;   // <-  link value for tuple found
        }  TuplePB;

        struct
        {
            UInt16    tupleDataMax; //  -> maximum size of tuple data area
            UInt16    tupleDataLen; // <-  number of bytes in tuple body
            TupleBody tupleData;    // <-  tuple data
        }  TupleDataPB;
    } u;
};
```

For the field values of `attributes`, see "CSGetFirstTuple" on page 49.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSInUseErr`	Card is configured and being used by another client
`kCSReadFailureErr`	Card cannot be read
`kCSBadCISErr`	Card Services has encountered a bad CIS structure
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSNoMoreItemsErr`	There are no more tuples to process

CSGetTupleData

This routine returns the contents of the last tuple returned by either `kCSGetFirstTuple`, or `kCSGetNextTuple`. Data returned is packed so that the tuple data is contiguous. The argument packet contains the same fields as `kCSGetFirstTuple`, or `kCSGetNextTuple`:

- The Socket field identifies the logical socket containing the PC Card.
- Attributes, DesiredTuple, Flags, LinkOffset, and CISOffset fields are for the internal use of Card Services, and the must contain the same values as `kCSGetFirstTuple`, or `kCSGetNextTuple`. Attributes and DesiredTuple describe the tuple being processed. Flags, LinkOffset, and CISOffset maintain state information during CIS processing requests.

```
pascal OSErr CSGetTupleData(GetTuplePB *pb)

typedef struct GetTuplePB GetTuplePB;
struct GetTuplePB
{
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> bitmap of attributes
   UInt8  desiredTuple;//  -> desired tuple code value, or $FF for all
   UInt8  tupleOffset; //  -> offset into tuple from link byte
   UInt16 flags;       // <-> internal use
   UInt32 linkOffset;  // <-> internal use
   UInt32 cisOffset;   // <-> internal use

   union
   {
      struct
      {
         UInt8  tupleCode;   // <-  tuple code found
         UInt8  tupleLink;   // <-  link value for tuple found
      }  TuplePB;

      struct
      {
         UInt16    tupleDataMax; //  -> maximum size of tuple data area
         UInt16    tupleDataLen; // <-  number of bytes in tuple body
         TupleBody tupleData;    // <-  tuple data
      }  TupleDataPB;
   } u;
};

// 'attributes' field values
enum
{
   kCSReturnLinkTuples = 0x0001
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request

Card and Socket Status

There is only one card and socket status routine, `CSGetStatus.`

CSGetStatus

This routine returns the following information about the current status of the PC Card and its socket:

- Identifies the logical socket in which the card is installed.
- Provides information about the card itself
  - whether it is write protected
  - if it is locked in the socket
  - whether a request has been made to insert the card into the socket or to eject it
  - the condition of the battery
  - the status of the card's readiness for an access
  - whether a card is actually present in the socket.
- It indicates any changes in the status of the card, as listed above.

```
pascal OSErr CSGetStatus(GetStatusPB *pb)

typedef struct GetStatusPB GetStatusPB;

struct GetStatusPB
{
   UInt16 socket;      //  -> logical socket number
   UInt16 cardState;   // <-  current state of installed card
   UInt16 socketState; // <-  current state of the socket
};
```

```
// 'cardState' field values

enum
{
   kCSWriteProtected= 0x0001,
   kCSCardLocked    = 0x0002,
   kCSEjectRequest  = 0x0004,
   kCSInsertRequest = 0x0008,
   kCSBatteryDead   = 0x0010,
   kCSBatteryLow    = 0x0020,
   kCSReady         = 0x0040,
   kCSCardDetected  = 0x0080
};

// 'socketState' field values

enum
{
   kCSWriteProtectChanged  = 0x0001,
   kCSCardLockChanged      = 0x0002,
   kCSEjectRequestPending  = 0x0004,
   kCSInsertRequestPending= 0x0008,
   kCSBatteryDeadChanged   = 0x0010,
   kCSBatteryLowChanged    = 0x0020,
   kCSReadyChanged         = 0x0040,
   kCSCardDetectChanged    = 0x0080
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified

Access Window Management

A window in this context is the block of system memory space assigned to the PC Card. The routines described in this section allow you to:

- Assign a window.
- Modify the characteristics of the window.
- Release the block of system memory space allocated.

CSRequestWindow

This routine assigns memory space, sets the base address and memory window size, defines access speed, and maps the logical socket number to the system memory address space.

```
pascal OSErr CSRequestWindow(ReqModRelWindowPB *pb)

typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
   UInt32 clientHandle;//  -> clientHandle returned by RegisterClient
   UInt32 windowHandle;// <-> window descriptor
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> window attributes (bitmap)
   UInt32 base;        // <-> system base address
   UInt32 size;        // <-> memory window size
   UInt8  accessSpeed; //  -> window access speed (bitmap)
                       //     (not applicable for I/O mode)
   UInt8  padding[1];
};

// 'attributes' field values

enum
{
   kCSMemoryWindow     = 0x0001,
   kCSIOWindow         = 0x0002,
   kCSAttributeWindow  = 0x0004, // not normally used by Card Services
                                 // clients
   kCSWindowTypeMask      = 0x0007,
   kCSEnableWindow        = 0x0008,
   kCSAccessSpeedValid    = 0x0010,
   kCSSwapLittleToBigEndian= 0x0020,// configure socket for
                                    little-endianess
   kCS16BitDataPath       = 0x0040,
   kCSWindowPaged         = 0x0080,
   kCSWindowShared        = 0x0100,
   kCSWindowFirstShared   = 0x0200,
   kCSWindowProgrammable  = 0x0400
};

// 'accessSpeed' field values

enum
{
   kCSDeviceSpeedCodeMask  = 0x07,
```

```
    kCSSpeedExponentMask   = 0x07,
    kCSSpeedMantissaMask   = 0x78,
    kCSUseWait             = 0x80,

    kCSAccessSpeed250nsec  = 0x01,
    kCSAccessSpeed200nsec  = 0x02,
    kCSAccessSpeed150nsec  = 0x03,
    kCSAccessSpeed100nsec  = 0x04,

    kCSExtAccSpeedMant1pt0 = 0x01,
    kCSExtAccSpeedMant1pt2 = 0x02,
    kCSExtAccSpeedMant1pt3 = 0x03,
    kCSExtAccSpeedMant1pt5 = 0x04,
    kCSExtAccSpeedMant2pt0 = 0x05,
    kCSExtAccSpeedMant2pt5 = 0x06,
    kCSExtAccSpeedMant3pt0 = 0x07,
    kCSExtAccSpeedMant3pt5 = 0x08,
    kCSExtAccSpeedMant4pt0 = 0x09,
    kCSExtAccSpeedMant4pt5 = 0x0A,
    kCSExtAccSpeedMant5pt0 = 0x0B,
    kCSExtAccSpeedMant5pt5 = 0x0C,
    kCSExtAccSpeedMant6pt0 = 0x0D,
    kCSExtAccSpeedMant7pt0 = 0x0E,
    kCSExtAccSpeedMant8pt0 = 0x0F,

    kCSExtAccSpeedExp1ns   = 0x00,
    kCSExtAccSpeedExp10ns  = 0x01,
    kCSExtAccSpeedExp100ns = 0x02,
    kCSExtAccSpeedExp1us   = 0x03,
    kCSExtAccSpeedExp10us  = 0x04,
    kCSExtAccSpeedExp100us = 0x05,
    kCSExtAccSpeedExp1ms   = 0x06,
    kCSExtAccSpeedExp10ms  = 0x07
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSBadBaseErr`	Invalid base address
`kCSBadAttributeErr`	Invalid window attributes

DIVERGENCE FROM PCMCIA STANDARD

Apple has added another attribute (`kCSIOTypeWindow`) that lets a client request that its new window be an I/O cycle window. Speed characteristics of an I/O window are fixed and any speed-related parameters are ignored. Speed parameters are considered only if the window is of the type `Memory` or `Attribute`.

In the PCMCIA standard, there is an implied window assignment when a client calls `RequestConfiguration` because the client must have called `RequestI/O` first. This assures the client that there is I/O window support for the change.

CSModifyWindow

This routine modifies the attributes and access speed of the window that were allocated by `CSRequestWindow`.

```
pascal OSErr CSModifyWindow(ReqModRelWindowPB *pb)

typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
   UInt32 clientHandle;//  -> clientHandle returned by RegisterClient
   UInt32 windowHandle;// <-> window descriptor
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  -> window attributes (bitmap)
   UInt32 base;        // <-> system base address
   UInt32 size;        // <-> memory window size
   UInt8  accessSpeed; //  -> window access speed (bitmap)
                       //     (not applicable for I/O mode)
   UInt8  padding[1];
};
```

For the field values of `attributes` and `accessSpeed`, see "CSRequestWindow" on page 55.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSBadBaseErr`	Invalid base address
`kCSBaddAttributeErr`	Invalid window attributes

DIVERGENCE FROM PCMCIA STANDARD

The `CSModifyWindow` routine must have a valid client handle (the one passed in on `CSRequestWindow`), otherwise a `kCSBadHandleErr` error is returned.

CSReleaseWindow

This routine releases the block of system memory assigned by `CSRequestWindow`.

```
pascal OSErr CSReleaseWindow(ReqModRelWindowPB *pb)

typedef struct ReqModRelWindowPB ReqModRelWindowPB;
struct ReqModRelWindowPB
{
   UInt32 clientHandle;//  -> clientHandle returned by RegisterClient
   UInt32 windowHandle;//  -> window descriptor
   UInt16 socket;      //  -> logical socket number
   UInt16 attributes;  //  not used
   UInt32 size;        //  not used
   UInt8  accessSpeed; //  not used
   UInt8  padding[1];  //  not used
};
```

For the field values of `attributes` and `accessSpeed`, see "CSRequestWindow" on page 55.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSBadHandleErr`	Invalid client handle

DIVERGENCE FROM PCMCIA STANDARD

`CSReleaseWindow` must have a valid client handle (the one passed in on `CSRequestWindow`), otherwise `kCSBadHandleErr` is returned.

Client Registration

When a PC Card is installed in a socket, it must be registered with Card Services. This allows Card Services to provide the client with event notifications, and to keep track of the clients that can manipulate PC Cards, using the client services routines described in this section. These routines allow you to:

- Get information about Card Services.
- Register clients.
- Deregister clients.

CSGetCardServicesInfo

This routine allows a client to detect the presence of Card Services. It returns the number of logical sockets installed; indicates whether Card Services is installed; and provides information about the vendor, such as revision number.

```
pascal OSErr CSGetCardServicesInfo(GetCardServicesInfoPB *pb)

typedef struct GetCardServicesInfoPB GetCardServicesInfoPB;
struct GetCardServicesInfoPB
{
   UInt8  signature[2];   // <-  two ascii chars 'CS'
   UInt16 count;          // <-  total number of sockets installed
   UInt16 revision;       // <-  BCD
   UInt16 kCSLevel;       // <-  BCD
   UInt16 reserved;       //  -> zero
   UInt16 vStrLen;        // <-> in: client's buffer size
                                out: vendor string length
   UInt8  *vendorString;  // <-> in: pointer to buffer to hold CS vendor
                          //         string (zero-terminated)
                          //     out: CS vendor string copied to buffer
};
```

RESULT CODES

`noErr`	No error

CSRegisterClient

Clients invoke this routine to make Card Services aware of their presence and to indicate their interest in various events. They also use the routine to indicate whether they are a memory or I/O client device driver, or a Memory Technology Driver (MTD).

```
pascal OSErr CSRegisterClient(RegisterClientPB *pb)

typedef struct RegisterClientPB RegisterClientPB;
struct RegisterClientPB
{
   UInt32          clientHandle; // <- client descriptor
   PCCardCSClientUPPclientEntry; // -> UPP to client's event handler
   UInt16          attributes;   // -> bitmap of client attributes
   UInt16          eventMask;    // -> bitmap of events to notify client
   Ptr             clientData;   // -> pointer to client's data
   UInt16          version;      // -> Card Services version client expects
};

// 'attributes' field values (see GetClientInfo)

// kCSMemoryClient              = 0x0001,
// kCSIOClient                  = 0x0004,
// kCSShareableCardInsertEvents  = 0x0008,
// kCSExclusiveCardInsertEvents  = 0x0010
```

SUPPLEMENTARY INFORMATION

Observe these precautions when using `CSRegisterClient`:

- It must not be called at interrupt time.
- You must specify the type of client for event notification order.
- You must set the event mask for types of events in which the client is interested. The event mask passed in during this call will be set for the global mask and all socket event masks.

RESULT CODES

`noErr`	No error
`kCSOutOfResourceErr`	Card Services lacks the resources to complete this request
`kCSBadAttributeErr`	Invalid window attributes

DIVERGENCE FROM PCMCIA STANDARD

The `CSRegisterClient` routine is synchronous. On returning from `CSRegisterClient` the client handle field is valid. Once this call is successful, all clients should support reentrancy. After calling `CSRegisterClient`, clients normally call `CSVendorSpecific` with `vsCode` set to `vsEnableSocketEvents`.

CSDeregisterClient

Clients invoke this routine when they want to remove themselves from the system. They must return all resources requested before calling this routine.

```
pascal OSErr CSDeregisterClient(RegisterClientPB *pb)

typedef struct RegisterClientPB RegisterClientPB;
struct RegisterClientPB
{
   UInt32          clientHandle; // <- client descriptor
   PCCardCSClientUPPclientEntry; // -> UPP to client's event handler
   UInt16          attributes;   // -> bitmap of client attributes
   UInt16          eventMask;    // -> bitmap of events to notify client
   Ptr             clientData;   // -> pointer to client's data
   UInt16          version;      // -> Card Services version client expects
};
```

For the field values of `attributes`, see “CSRegisterClient” on page 60.

RESULT CODES

`noErr`	No error
`kCSBadHandleErr`	Invalid client handle
`kCSBadAttributeErr`	Invalid window attributes

Miscellaneous Routines

The routines described in this section help you with various Card Services management tasks, such as:

- Resetting the PC Card.
- Validating the CIS (Card Information Structure).
- Getting vendor information that is specific to Apple Computer, Inc.

CSResetCard

This routine resets the logical socket number. It can also reset PC Card attributes, but this function of the routine is not used in this application.

```
pascal OSErr CSResetCard(ResetCardPB *pb)

typedef struct ResetCardPB ResetCardPB;
struct ResetCardPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  socket;        //  -> socket number
   UInt16  attributes;    //  not used
};
```

SUPPLEMENTARY INFORMATION

Calling clients will receive `kCSResetCompleteMessage` regardless of whether or not their socket event mask and global event mask have set `kCSResetEvent`.

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSBadHandleErr`	Invalid client handle

DIVERGENCE FROM PCMCIA STANDARD

Card Services does not issue `kCSCardResetMessage` in place of `kCSCardReadyMessage`. If a client is issuing a reset to a card, then it should know whether the card will generate a `kCSCardReadyMessage` or not. If the card goes through a transition from busy to ready, then the client will know that it should not access the card until it receives the `kCSCardReadyMessage` event.

CSValidateCIS

This routine validates the CIS on the PC Card in the socket specified. It identifies the logical socket that contains the PC Card and returns the number of valid tuple chains located in the CIS.

```
pascal OSErr CSValidateCIS(ValidateCISPB *pb)

typedef struct ValidateCISPB ValidateCISPB;
struct ValidateCISPB
{
   UInt16  socket;     //  -> socket number
   UInt16  chains;     //  -> whether link/null tuples should be included
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketErr`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSBadCISErr`	Card Services has detected a bad CIS

DIVERGENCE FROM PCMCIA STANDARD

The PCMCIA standard indicates that a `kCSBadCISErr` result should be returned by setting the `pb->chains` element to zero. To accomodate cards that have no tuples, Apple returns `kCSBadCISErr` as a result code if the CIS is bad. If `noErr` is returned, then the value in the `pb->chains` reflects the number of valid tuples, not counting link tuples.

CSVendorSpecific

`CSVendorSpecific` is defined to allow Apple to extend the interface definition of Card Services for elements that are specific to the Mac OS. The call requires two paramenters, `clientHandle` and `vsCode`.

```
pascal OSErr CSVendorSpecific(VendorSpecificPB *pb)

typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  vsCode;
   UInt16  socket;
```

```
   UInt32  dataLen;       //  -> length of buffer pointed to by
vsDataPtr
   UInt8   *vsDataPtr;    //  -> Card Services version this client
expects
};

//  'vsCode' field values

enum
{
   vsAppleReserved        = 0x0000,
   vsEjectCard            = 0x0001,
   vsGetCardInfo          = 0x0002,
   vsEnableSocketEvents   = 0x0003,
   vsGetCardLocationIcon  = 0x0004,
   vsGetCardLocationText  = 0x0005,
   vsGetAdapterInfo       = 0x0006
};
```

SUPPLEMENTARY INFORMATION

Additional parameters may be required for each vendor-specific code, as described in the following sections. The parameters that may be required are:

- `EjectCard Parameter Block`
- `GetCardInfo Parameter Block`
- `EnableSocketEvents Parameter Block`
- `GetAdapterInfo Parameter Block`

RESULT CODES

`noErr`	No error
`kCSUnsupportedFunctionErr`	Invalid vendor-specific code

EjectCard Parameter Block

If clients have configured their PC Cards themselves, they must pass in their client handle when they wish to eject such cards. Clients may not be able to eject cards they have not configured until the card is reconfigured.

```
// vendor-specific call #1

typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  vsCode;        //  -> vsCode = 1
```

```
   UInt16  socket;        //  -> desired socket number to eject
   UInt32  dataLen;       //  not used
   UInt8   *vsDataPtr;    //  not used
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketError`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket
`kCSInUseErr`	Card is configured and being used by another client

GetCardInfo Parameter Block

Calling this routine allows the client to get vendor-specific information, as detailed in the following code.

```
// vendor-specific call #2

typedef struct GetCardInfoPB GetCardInfoPB;
struct GetCardInfoPB
{
   UInt8  cardType;    // <-  type of card in socket (defined at top of
                          file)
   UInt8  subType;     // <-  detailed card type (defined at top of file)
   UInt16 reserved;    // <-> reserved (should be set to zero)
   UInt16 cardNameLen; //  -> maximum length of card name to be returned
   UInt16 vendorNameLen;// ->  maximum length of vendor name to be
                          returned
   UInt8  *cardName;   //  ->  ptr to card name string (read from CIS),
                          or nil
   UInt8  *vendorName; //  ->  ptr to vendor name (read from CIS), or nil
};

// GetCardInfo card types

#define  kCSUnknownCardType         0
#define  kCSMultiFunctionCardType   1
#define  kCSMemoryCardType          2

#define  kCSSerialPortCardType      3
#define  kCSSerialOnlyType          0
#define  kCSDataModemType           1
#define  kCSFaxModemType            2
#define  kCSFaxAndDataModemMask     (kCSDataModemType | kCSFaxModemType)
#define  kCSVoiceEncodingType       4
```

```
#define  kCSParallelPortCardType     4

#define  kCSFixedDiskCardType        5
#define  kCSUnknownFixedDiskType     0
#define  kCSATAInterface             1
#define  kCSRotatingDevice           (0<<7)
#define  kCSSiliconDevice            (1<<7)
#define  kCSVideoAdaptorCardType     6

#define  kCSNetworkAdaptorCardType   7
#define  kCSAIMSCardType             8
#define  kCSNumCardTypes             9
```

RESULT CODES

`noErr`	No error
`kCSBadSocketError`	Invalid socket specified
`kCSNoCardErr`	No card in the specified socket

EnableSocketEvents Parameter Block

Calling this routine is equivalent to calling the old `RequestSocket` mask for every socket in the system, using the global event mask as the starting socket event mask.

```
// vendor-specific call #3

typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
   UInt32 clientHandle;//  -> clientHandle returned by RegisterClient
   UInt16 vsCode;      //  -> vsCode = 3
   UInt16 socket;      //  not used
   UInt32 dataLen;     //  not used
   UInt8  *vsDataPtr;  //  not used
};
```

RESULT CODES

`noErr`	No error
`kCSBadHandleErr`	Invalid client handle

DIVERGENCE FROM PCMCIA STANDARD

This call is not a standard PCMCIA call. It provides a better way to enable events after reentrance into a client is available.

GetAdapterInfo Parameter Block

Socket Services API elements are frequently not brought out to the Card Services API but are still required for normal card operation. This call allows clients to query the capabilities of an adapter that interfaces to a given socket. This information may be used to improve the operation of a client with a given socket and card.

```
// vendor-specific call #6

typedef struct VendorSpecificPB VendorSpecificPB;
struct VendorSpecificPB
{
   UInt32  clientHandle;  //  -> clientHandle returned by RegisterClient
   UInt16  vsCode;        //  -> vsCode = 6
   UInt16  socket;        //  -> socket number
   UInt32  dataLen;       //  -> length of GetAdapterInfoPB plus space for
                          //     voltages
   UInt8   *vsDataPtr;    //  -> GetAdapterInfoPB * (supplied by client)
};

typedef struct GetAdapterInfoPB GetAdapterInfoPB;

struct GetAdapterInfoPB
{
   UInt32  attributes;    // <-  capabilties of socket's adapter
   UInt16  revision;      // <-  id of adapter
   UInt16  reserved;      //
   UInt16  numVoltEntries;// <-  number of valid voltage values
   UInt8   *voltages;     // <-> array of BCD voltage values
};



// 'attributes' field values

enum
{
   kCSLevelModeInterrupts     = 0x00000001,
   kCSPulseModeInterrupts     = 0x00000002,
   kCSProgrammableWindowAddr  = 0x00000004,
   kCSProgrammableWindowSize  = 0x00000008,
   kCSSocketSleepPower        = 0x00000010,
   kCSSoftwareEject           = 0x00000020,
   kCSLockableSocket          = 0x00000040,
   kCSInUseIndicator          = 0x00000080
};
```

RESULT CODES

`noErr`	No error
`kCSBadSocketError`	Invalid socket specified

Unsupported Routines

`CSRequestExclusive` and `CSReleaseExclusive` are not supported by the PowerBook Card Services API.

PC Card Manager Constants

The PC Card Manager helps client software to recognize, configure, and view PC Cards that are installed in the PC Card sockets on PowerBook computers. This section lists the PC Card Manager constants and explains the function of each constant.

```
// miscellaneous

#define  CS_MAX_SOCKETS  32   // a long is used as a socket bitmap

enum
{
   gestaltCardServicesAttr    = 'pccd', // Card Services attributes
   gestaltCardServicesPresent = 0       // if set, Card Services is present
};

enum
{
   _PCCardDispatch  = 0xAAF0  // Card Services entry trap
};

/*
   The PC Card Manager will migrate towards a complete Macintosh name
space very soon. Part of that process will be to reassign result codes
to a range reserved for the PC Card Manager. The range will be -9050 to
-9305 (decimal inclusive).
*/
```

```
// result codes
enum
{

kCSBadAapterErr             = -9050    // invalid adapter number
kCSBadAttributeErr          = -9051    // attributes field value is invalid
kCSBadBaseErr               = -9052    // base system memory address is invalid
kCSBadEDCErr                = -9053    // EDC generator specified is invalid
kCSBadIRQErr                = -9054    // specified IRQ level is invalid
kCSBadOffsettErr            = -9055    // specified PC Card memory array offset is
                                       // invalid
kCSBadPageErr               = -9056    // specified page is invalid
kCSBadSizeErr               = -9057    // specified size is invalid
kCSBadSocketErr             = -9058    // specified logical or physical socket
                                       // number is invalid
kCSBadTypeErr               = -9059    // specified window or interface type is
                                       // invalid
kCSBadVccErr                = -9060    // specified Vcc power level index is invalid
kCSBadVppErr                = -9061    // specified Vpp1 or Vpp2 power level index
                                       // is invalid
kCSBadWindowErr             = -9062    // specified window is invalid
kCSBadArgLengthErr          = -9063    // ArgLength argument is invalid
kCSBadArgsErr               = -9064    // values in argument packet are invalid
kCSBadHandleErr             = -9065    // clientHandle is invalid
kCSBadCISErr                = -9066    // CIS on card is invalid
kCSBadSpeedErr              = -9067    // specified speed is unavailable
kCSReadFailureErr           = -9068    // unable to complete read request
kCSWriteFailureErr          = -9069    // unable to complete writer request
kCSGeneralFailureErr        = -9070    // an undefined error has occurred
kCSNoCardErr                = -9071    // no PC Card in the socket
kCSUnsupportedFunctionErr   = -9072    // function is not supported by this
                                       // implementation
kCSUnsupportedModeErr       = -9073    // mode is not supported
kCSBusyErr                  = -9074    // unable to process request at this time
kCSWriteProtectedErr        = -9075    // media is write-protected
kCSConfigurationLockedErr   = -9076    // a configuration has already been locked
kCSInUseErr                 = -9077    // resource is being used by a client
kCSNoMoreItemsErr           = -9078    // there are no more of the items requested
kCSOutOfResourceErr         = -9079    // Card Services has exhausted the resource

};
```

```
        // messages sent to client's event handler
        enum
        {
kCSNullMessage                  = 0x00    // no messages pending (not sent to clients)
kCSCardInsertionMessage         = 0x01    // card has been inserted into the socket
kCSCardRemovalMessage           = 0x02    // card has been removed from the socket
kCSCardLockMessage              = 0x03    // card is locked into the socket with a
                                          // mechanical latch
kCSCardUnlockMessage            = 0x04    // card is no longer locked into the socket
kCSCardReadyMessage             = 0x05    // card is ready to be accessed
kCSCardResetMessage             = 0x06    // physical reset has completed
kCSInsertionRequestMessage      = 0x07    // request to insert a card using insertion
                                          // motor
kCSInsertionCompleteMessage     = 0x08    // insertion motor has finished inserting a
                                          // card
kCSEjectionRequestMessage       = 0x09    // user or other client is requesting a card
                                          // ejection
kCSEjectionFailedMessage        = 0x0A    // eject failure due to electrical or
                                          // mechanical problems
kCSPMResumeMessage              = 0x0B    // power management resume
kCSPMSuspendMessage             = 0x0C    // power management suspend
kCSResetPhysicalMessage         = 0x0D    // physical reset is about to occur
kCSResetRequestMessage          = 0x0E    // client has requested physical reset
kCSResetCompleteMessage         = 0x0F    // ResetCar() background reset has completed
kCSBatteryDeadMessage           = 0x10    // battery is no longer usable; data will be
                                          // lost
kCSBatteryLowMessage            = 0x11    // battery is weak and should be replaced
kCSWriteProtectMessage          = 0x12    // card is now write protected
kCSWriteEnableMessage           = 0x13    // card is now write enabled
kCSClientInfoMessage            = 0x14    // client is to return client information
kCSSSUpdatedMessage             = 0x15    // AddSocketServices/ReplaceSocketServices
                                          // has changed SS (SocketServices) support
kCSFunctionInterruptMessage     = 0x16    // card function interrupt
kCSAccessErrorMessage           = 0x17    // client bus made error on access to socket
kCSCardUnconfiguredMessage      = 0x18    // a kCSCardReadyMessage was delivered to
                                          // all clients and no clients request a
                                          // configuration for the socket
kCSStatusChangedMessage         = 0x19    // status change for cards in I/O mode

        };
```

CHAPTER 4

Device Drivers

This chapter provides guidelines for developers of PC Card device drivers for PowerBook computers. It describes questions that commonly come up during development and suggests answers to those questions.

Driver Loading

Currently there is no defined mechanism for installing and loading drivers from PC Cards. Apple is working on an architecture for loading device drivers from PC Cards, but neither the PCMCIA committee nor the Apple development team has a solution at this time. Later versions of this developer note will describe any driver loading architecture that Apple develops before the first product release.

In the first system release, all drivers and other client software are stored in the PC Card's expansion ROM. Because the expansion ROM is electronically erasable, its contents can be changed in the field if necessary.

Drivers may also be loaded from a disk or other source by the Macintosh system software.

Booting Requirements

The hierarchical file system (HFS) storage driver lets the HFS mount data storage and search for a bootable system. To boot the Macintosh system from a PC Card, the HFS storage driver must be

- aware of the Slot Manager. It must know how to install a DCE (device control entry) that is compatible with booting from a slot device.
- present in the PC Card's expansion ROM or in the Macintosh ROM.

Guidelines for Socket Developers

Card Services supports the process of adding and deleting Socket Services modules. Socket Services modules are developed to enable support for different socket adapters or different adapter topologies. For example, a PCMCIA adapter (controller) may be added to a system via a PDS (processor-direct slot) connector and therefore requires a version of Socket Services that knows how to handle interrupts from the PDS adapter.

Card Services provides `AddSocketServices` for Socket Services developers. When `AddSocketServices` is called, the Socket Services to be installed passes an entry point, a unique (usually version-associated) ID, the number of adapters and sockets supported by the installing Socket Services module, and a pointer to the Socket Services globals.

Card Services passes the installed Socket Services globals pointer to it during each function call made to Socket Services. The `AddSocketServices` parameter block structure is as follows:

```
typedef struct
{
   Ptr      SSEntry;       // entry point to SS
   ushort   Attributes;    // unique id
   ushort   NumAdapters;   // number of adapters supported
   ushort   NumSockets;    // number of sockets supported
   Ptr      DataPtr;       // pointer to SS globals
}
AddSocketServicesPB;
```

Interrupt Support

Interrupt support for drivers is handled at multiple levels within the Mac OS architecture. When a client registers with Card Services, the client passes a client callback address that Card Services stores for all callbacks to the client. At the same time, the registering client also passes an event mask for all events for which it wants to receive callback notification. A client can adjust the event mask at some later time with `GetClientEventMask` and `SetClientEventMask`.

When an event generated by Card Services or Socket Services is destined for a client (for example, if the client has previously indicated that it wants to be notified of certain events on a given socket) Card Services uses the appropriate event callback interface described in the Card Services section of the *PCMCIA Standards*. There are different callback arguments based on the event type.

Some events are artificially generated by Card Services. For example, when a client registers after a PC Card has already been inserted into a socket, Card Services generates a `kCSCardInsertionMessage` event for the newly registered client. In this way, clients can be designed to do PC Card tasks in response to event callbacks. These artificial events execute at interrupt level 0.

Other events may be the result of an interrupt generated by the Socket Services adapter. An example of this is a `kCSFunctionInterruptMessage` event, which would be generated by an adapter whose socket has a PC Card that asserts `–IREQ`. This type of event is dispatched to the client at the interrupt level at which the adapter interrupt came in. In this case the client has to be aware that any event-handling code must operate within the bounds of the normal execution restrictions placed on Macintosh interrupt time. Refer to *Inside Macintosh: Memory* for further information.

Alternative PCMCIA Controllers

The PowerBook Card Services architecture is designed to support alternative PCMCIA controllers in conformance with the intent of the PCMCIA standard. The PC Card support architecture includes the ability to substitute PCMCIA controllers (and provide an accompanying Socket Services module) and have existing clients work with the new controller. There are no architectural barriers that would prevent Apple from adopting other PCMCIA controllers in the future.

CHAPTER 5

Human Interface

This chapter discusses some of the human interface issues that are important to developers designing panels or developing software for PC Cards in the PowerBook environment.

The PCMCIA standard supports mass storage cards as well as I/O cards, such as modems, network cards, and video cards. The Macintosh desktop metaphor already includes the concept of storage device representation (for example, floppy disks, hard disks, servers, and CD-ROMs) so it automatically supports PC Cards that provide mass storage. Since users are already familiar with manipulating desktop icons for these storage devices, Apple has extended the metaphor to include I/O cards as well. This approach has the following advantages:

- It provides a more consistent user experience for all types of PC Cards.
- It informs the user that the card is installed.
- It provides greater protection for the user, the card, and the operating system by providing a software-controlled removal mechanism for all cards.

Manual Card Ejection

PowerBook computers currently support PC Card ejection using a software command. Ejection is controlled by Card Services which can eject a PC Card after notifying all card clients that the card is about to be ejected. If clients are using resources on the card, they have the option of refusing the request and telling users why the card cannot be ejected.

In the future, software ejection may not be the norm. Software clients may have to deal with the situation where a PC Card is removed without notification. Currently, a user may manually eject the card in an emergency by inserting a paper clip into a hole near the card socket. (This method of ejection is similar to the method of emergency ejection used with floppy diskettes in a floppy disk drive.) For this reason, clients must be careful if they access PC Card addresses (registers and memory locations) directly because they cannot rely on advice from Card Services before the PC Card is ejected, and clients it must be aware that the access may fail under these conditions.

A mechanism built into Card Services prepares for and handles unexpected PC Card removal. When Card Services detects an access error to a PC Card it sends an access error message to each registered PC Card client. Clients may want to set an internal flag and halt access to the card during the next attempt to access the card.

Finder Extension

Support for I/O-oriented PC Cards is provided through a Macintosh **Finder extension** that is a client of Card Services. The Finder extension mechanism was chosen because it is the only external means of providing access to the Finder's internal code. The extension maintains card icons on the desktop, provides custom card information in Get

Info windows, and ejects cards when they are dragged to the Trash folder. The Finder extension also helps a client to provide custom icons, card names, card types, help messages, and other custom features (based on card type) when a card is opened. For example, a typical custom feature would be to open the Monitors control panel when the user double-clicks a video card's icon.

The following sections describe various Finder extension support features.

Card Services Client Registration

The Finder extension registers itself with Card Services as part of its startup process. After that point, it simply tracks card events to determine when cards have been inserted or removed. It ignores storage cards because they are normally handled by device drivers.

The client event handler and extension code communicate with each other using shared global variables. This is necessary because the event handler receives card events at interrupt time. The shared variables let the extension code process events at a later (noninterrupt) time, when it can use all parts of the Macintosh Toolbox.

Card Icons

Macintosh users benefit from having icons and names that reflect the functionality or the type of the card, and each installed PC Card is identified by a custom icon. The name associated with the icon is the name taken from the card's level 1 version tuple. If the tuple is missing or no name is specified, the card is assigned the name "Untitled." The user cannot change the card's icon or name. Figure 5-1 shows a sample PC Card icon.

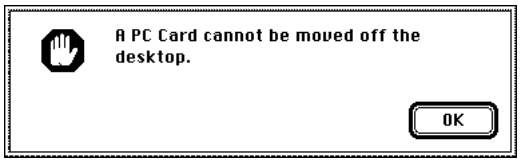
Figure 5-1 Sample PC Card icon

Client software can provide a custom icon, overriding the default icon for the card type. It may also override the card's name. See "Custom Support for I/O Cards" on page 81 for more information about overriding card icons and names.

User Interactions

When a PC Card is inserted, the Finder extension places its icon on the Macintosh desktop. At this point the icon can be dragged anywhere on the desktop or placed in the Trash folder. If the user attempts to drag the icon to a folder or disk icon, or to an open folder or disk window, the user is presented with a dialog box (Figure 5-2) that indicates the icon must remain on the desktop, and the icon returns to its previous location.

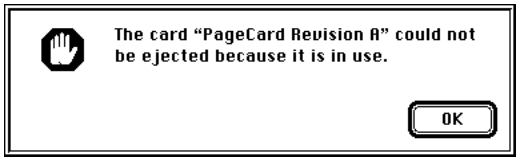
Figure 5-2 Icon dragging warning



If the user drags a PC Card icon onto an application's icon and the application starts up, the card's file will not be included in the list of files sent to the application for processing.

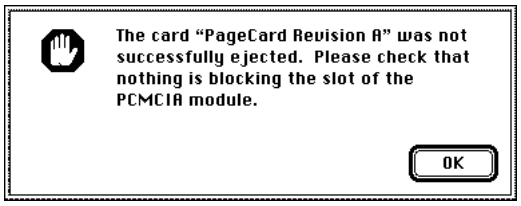
If the user drags a card icon to the Trash folder, the Finder extension tries to eject the card. If the card is in use, the ejection operation fails and the user is presented with the dialog box shown in Figure 5-3.

Figure 5-3 Card ejection warning



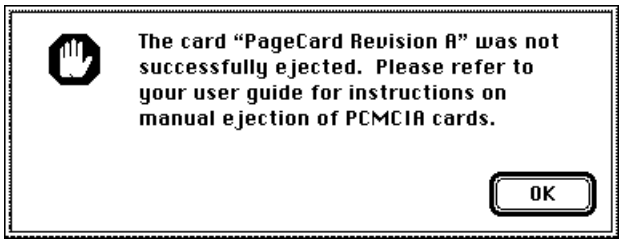
If the card cannot be ejected because of a failure of the ejection mechanism or because the card's slot is blocked, the user is presented with a dialog box that describes the problem and indicates what to do about it. Figure 5-4 shows atypical ejection failure dialog box.

Figure 5-4 Ejection failure warning



If the user attempts to eject the card a second time, and this attempt also fails, the user is presented with a dialog box pointing to the user guide for instructions on manual ejection of the card. Figure 5-5 shows this dialog box.

Figure 5-5 User guide reference warning

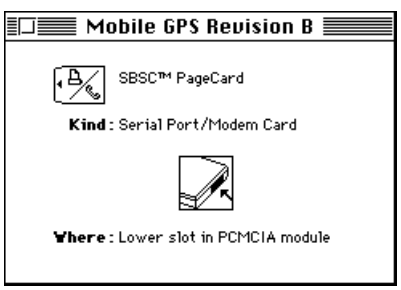


When a card has been successfully ejected, the Finder extension removes its icon from the desktop. It also closes the card's Get Info window if it was open.

Card Information Display

Most Finder objects (disks, files, folders, Trash folder) are allowed to display information about themselves using a Get Info window. PC Cards are no exception. The Get Info windows for PC Cards contain all relevant information about the PC Cards such as their icon, name, function, and location. Figure 5-6 shows an example of a PC Card Get Info window.

Figure 5-6 Sample PC Card Get Info window

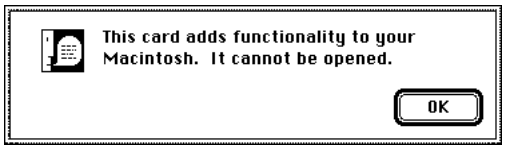


Custom Card Actions

A PowerBook user who double-clicks on an icon or selects Open from the File menu expects something to happen. Typically, the item represented by the icon opens. Apple does not currently implement any standard icon opening behavior for PC Cards. However, Card Services and the Finder extension let developers supply custom actions.

Since many I/O cards have no user interface elements, opening a card may be meaningless. In this case, double-clicking on the desktop icon displays a dialog box that names the card and gives a generic message about it, as shown in Figure 5-7.

Figure 5-7 Generic message for cards that cannot be opened



Card Services provides a mechanism that lets clients define custom actions for specific card types. The Finder extension uses this mechanism to override the default open action by first asking the client to perform a custom action. If no custom action is defined, the Finder extension executes its default action.

Some examples of custom actions that a client might perform include

- helping the user select and open a terminal program for modem cards
- opening an address book application for a pager card

When defining custom actions, it should be easy to access a card's associated elements (such as a page card's address book application) when you double-click the PC Card's icon. If some elements do not open immediately, a dialog box is displayed that directs you to the interface element requiring attention.

IMPORTANT
Low-level PC Card support software should not implement user interface actions. However, low-level software may initiate events to be handled by higher-level software. ▲

Software Not Installed

When you are using PC Cards, you generally need specific application software. For example, you need networking software for the LAN cards, and so forth. If you install a card and the application software is not installed on your computer, you will see the type of message shown in Figure 5-8. You can choose to ignore the message by clicking Cancel, or eject the PC Card by clicking Eject. You will not be able to access the card until you install the appropriate software.

Figure 5-8 Missing software warning

Custom Support for I/O Cards

The Finder extension provides a mechanism for developers that supports custom icons, names, types, help messages, actions, and other custom features. Observe the following guidelines when customizing these elements of PC Cards:

- You can design custom icons that are passed to the Macintosh Finder. The custom icon should represent the functionality of the card and look similar to Apple PC Card icons. The shape of the icon should be the same as the shape of Apple's icons, although you may use a unique symbol or logo inside this shape to identify the card as coming from a particular developer.
- Card names provided by the card vendor may be overwritten with names provided by the software developer. These names should be placed in a resource so that they may be localized. For example, if a vendor supplies a card name "XY-5Y-22A," which is meaningless to the user, you may provide a card name such as "ACME Modem" to explain the functionality provided by the card.
- You may override card types defined by the Finder. The Finder displays the information to the user in the Kind field of the card's Get Info window. For example, the Finder may define a card type as "Serial Card" but you may override this with the more specific card type "FAX Modem Card."
- You may customize balloon help messages to provide more specific card information. For example, if the Finder includes the help message "This is a serial card…" you may substitute the message "This is a FAX Modem card…"
- Custom card actions are discussed in "Custom Card Actions" on page 79. You can defines custom card actions that will be performed when the user double-clicks on the desktop card icon. If you do not define custom actions, If it has no default action, it displays the generic message shown in Figure 5-7.

Multifunction Cards

A multifunction card is one that can perform at least two discrete functions, such as modem and network functions. This type of card is supported by the latest release of the PCMCIA standard, which is documented in *PC Card Standard*, February 1995, and referred to in this section as the "February release."

Apple Computer, Inc. provides full software support for PC Cards, as defined by the PCMCIA standard, release 2, and documented in *PCMCIA Standards*, Release 2.01, November 1992. This standard does not support multifunction cards and, if you use a multifunction card with the software currently supplied by Apple, only the first function on the card will be recognized by the software.

There are ways currently available to work around this situation. This section provides an overview of February-release support for multifunction cards, and indicates ways that developers may use other Apple resources to accommodate multifunction cards with the Apple software currently available.

IMPORTANT
Before you attempt to write drivers or other software to support multifunction cards, you should read the relevant sections in *PC Card Standard*, February 1995, and consult your technical support representative at Apple Computer, Inc. ▲

February-Release Support

Multifunction cards contain multiple Card Information Structures (CIS). The first CIS, which is shown in Figure 5-9 as the global CIS, identifies the card as being one that contains multiple functions. It does this by means of the `CISTPL_LONGLINK_MFC` tuple (MFC tuple). There are separate CIS for each of the functions supported, that is for each set of configuration registers on the card.

The MFC tuple performs the following functions:

- It provides the link to the next tuple.
- It indicates the number of sets of configuration registers (that is the number of functions implemented by the card)
- It provides the target addresses for each of the functions.

Table 5-1 summarizes the functions of the different bytes in the MFC tuple. You will find detailed information on this subject in *PC Card Standard*, February 1995, Volume 4.

When the client parses the MFC tuple, it finds the first tuple listed. In the example shown, this is the modem tuple. Subsequently the MFC tuple identifies and targets all other functions on the PC Card.

Figure 5-9 Parsing tuples for multifunction cards — February release

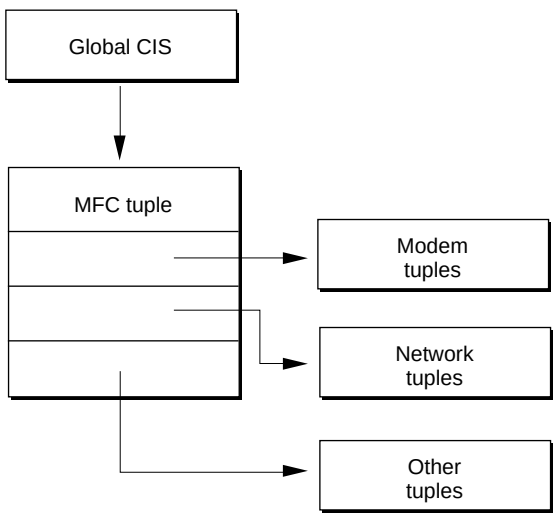


Table 5-1 MFC tuple functions

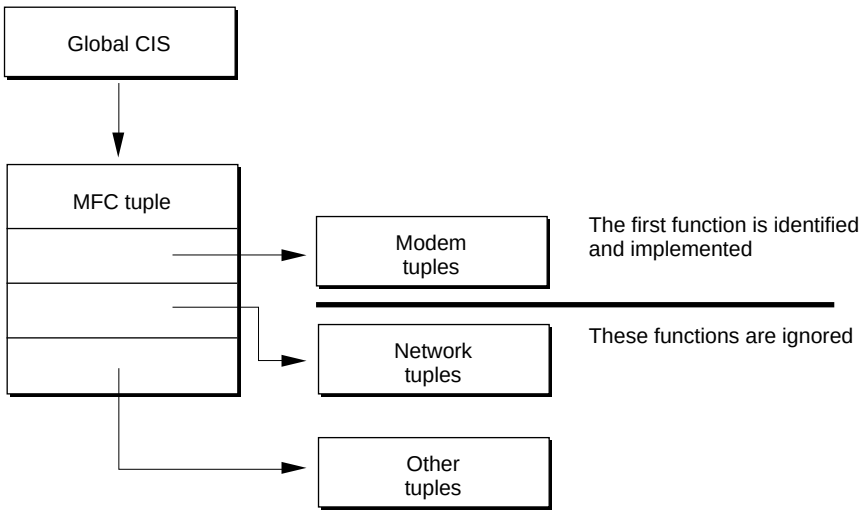
Byte number	Field	Description
0	TPL_CODE	CISTPL_LONGLINK_MFC
1	TPL_LINK	Link to next tuple. This will be at least byte 6, figuring 5 bytes per function.
2	TPLMFC_NUM	Number of sets of configuration registers. This gives the count of the number of functions on the card.
3	TPLMFC_TAS1	CIS target address space for the first function on the PC Card.
4-7	TPLMFC_ADDR1	Target address for the first function stored as an unsigned long integer, with the low-order byte first.
8	TPLMFC_TAS2	CIS target address space for the second function on the PC Card.
9-12	TPLMFC_ADDR2	Target address for the second function stored as an unsigned long integer, with the low-order byte first.
13-n		Additional target address space and address fields for any additional functions on the PC Card. If there are only two functions, these fields will not be present.

Release 2 Support

The MFC tuple is not supported by PCMCIA Standard Release 2 Card Services. Therefore the process of identifying multiple functions described in the previous section does not work with multifunction PC Cards currently used in Apple applications.

As shown in Figure 5-10, the global CIS identifies the first function (modem tuple), but will ignore any other tuples. This means that the multifunction card is actually being used as a single-function card.

Figure 5-10 Parsing tuples for multifunction cards — Release 2



It is possible to parse multiple tuples manually, using `GetFirstTuple` and `GetNextTuple`, combined with the return links attribute set, which is the Apple constant `kCSReturnLinkTuples`. This process involves writing a client driver that can call Card Services and parse the information it receives. The client driver gets the first tuple, and then proceeds through the CIS identifying the tuple for each card function in turn.

IMPORTANT

This developer note does not describe the process for manual parsing, and you should contact your Apple technical support representative if you wish to use multifunction cards with the Release 2 of the PCMCIA standard. ▲

Apple Computer, Inc. is conducting an ongoing investigation to establish rules for identifying and controlling multifunction PC Cards on the Macintosh desktop, and is currently considering updating the software supplied with hardware that accommodates multifunction PC Cards. There should be a desktop icon for each type of functionality provided on a card, but Apple has yet to establish the mechanism for supporting user actions for individual functions of a card.

The committee for PCMCIA standards is in the process of defining how Card Services, Socket Services, and interrupt-sharing should work with multifunction PC Cards.

Glossary

asynchronous This term is applied to processes and operations in which the sequencing of events is controlled by free-running signals. Each event is triggered by the completion of the previous event. The alternate type of operation is known as **synchronous.**

big-endian Data formatting in which each field is addressed by referring to its most significant byte. This means that if you are accessing a four-byte, 32-bit data word, the most significant byte is byte 03, and the most significant bit is bit 31. Macintosh computers use the big-endian data format. Computers based on Intel architectures, such as IBM PCs, use the little-endian format. See also **little-endian.**

Card Information Structure (CIS) A list of structures that describe the functions and capabilities of a PC Card. These structures are variable in length, and are made up of data blocks referred to in this context as **tuples.** The CIS is generally written only once, when the card is formatted.

Card Services The part of the PC Card Manager that provides system services for third-party PC Card control software.

client A device driver or application program that uses the Card Services software.

event handler A routine in a client that Card Services uses to notify the client of events. This routine lets clients of a particular PC Card handle interrupts from functions on the card.

Finder extension A client of Card Services. It is the only external means of accessing the Finder's internal code. The extension maintains card icons on the desktop, provides custom card information in the Get Info window, and ejects cards when they are dragged to the Trash. The Finder extension also helps clients provide custom icons, card names, card types, help messages, and other custom features.

glue routine A run-time library routine, usually provided by the development environment. It provides a linkage between a high-level language code and a system routine with an interface protocol different from that of the high-level language. It is also any short special-purpose assembly-language routine.

hardware abstraction This is a process that takes hardware functionality and gives it a name, thus concealing the hardware implementation from the software. The hardware abstraction layer acts as a liason between the software element and the hardware element.

JEDEC The Joint Electron Device Engineering Council is one of the groups that determines engineering standards in the United States.

little-endian Data formatting in which each field is addressed by referring to its least significant byte. This means that if you are accessing a four-byte, 32-bit word, the most significant byte is byte 00, and the most significant bit is bit 00. Computers based on Intel architectures, such as IBM PCs, use the little-endian format. Macintosh computers use the big-endian format. See also **big-endian.**

PC Card An expansion card that conforms to the PCMCIA standard, and may be inserted into a 68-pin socket in the PowerBook computer. PC Cards provide such functions as additional storage, fax/modem support, video support, and LAN (local area network) support.

PC Card Manager Part of the Mac OS that supports PC cards in PowerBook computers. The PC Card Manager helps client software to recognize, configure, and view PC Cards that are inserted into PC Card sockets on PowerBook computers.

PCMCIA controller The hardware interface to PC Cards. It provides the interface signals, configurable voltages to power the cards, hardware windows into the card's address space, and interrupt decoding for state changes.

PCMCIA standards An industry standard for computer expansion cards set by the Personal Computer Memory Card Internal Association.

pseudocode This is an algorithm that is not written in any real computer language. It is generally written in English, or in something close to a computer language.

reentrant A reentrant routine is one that is able to accept a call while one or more previous calls to it are pending. It can do this without invalidating the previous call(s).

socket The hardware receptacle into which a PC Card is inserted.

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

stubbed message A stub is a piece of code that has no function. A stubbed message, therefore, instead of handling some situation, generally returns without doing anything.

synchronous This term is applied to processes and operations in which the sequencing of events is controlled by clock pulses. The alternate type of operation is described as **asynchronous**.

tuple Tuples are elements of the **CIS**. They are blocks of data made up of eight-bit bytes. Each tuple contains information about itself, including its length, type, and information about the PC Card. Host software examines the tuples to determine the capabilities of the card, such as checksum control, **JEDEC** programming information, and configuration information. The tuple also contains the link to the next tuple, or an indicator showing that it is the last tuple in the list.

window The term window is used in this developer note to indicate the block of system memory space assigned to the PC Card. It is a defined address range that you can use to perform read or write accesses to the card Each card slot is assigned a block of memory. Each block of memory is divided into two sections. You can access the card in both sections at the same time, for example, you could do a buffer access using one section and generate an I/O cycle using the other section. This use of the term window should be distinguished from the standard Macintosh usage, where a window refers to some sort of panel displayed on the screen, such as the Get Info window.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z

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

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

WRITERS
Joyce D. Mann and George Towner

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATOR
Sandee Karr

PRODUCTION
Rex Wolf

Special thanks to Sue Bartalo, James Blair, Steve Carlton, Steve Christensen, Godfrey DiGiorgi, Dave Falkenberg, Jerry Katzung, Mike Primeau, D. K. Smith, and Carlton Van Putten