Mac OS System PC Card Family 3.0
Developers Guide

Version 1.5.1

Prepared By:

SystemSoft Corporation
&
Apple Computer, Inc.

Creation Date:February 26, 1996
Modified Date:January 8, 1997

1.0 Overview

The Macintosh PC Card Family architecture is a multi-layered architecture designed for robustness, extensibility and ease of maintenance. The layers are implemented as shared libraries, plug-ins or inits in order to make the best use of the Mac OS 7.5.x (and future versions of the Mac OS.)

PCCard 3.0 is designed to work within the Macintosh environment and is not a port of existing technology from other platforms. SystemSoft is applying what has been learned from years of PC Card support on Intel platforms to build an implementation that avoids the limitations of the DOS platform, but uses our experience with many vendors PC Cards and interface hardware. Our implementation is designed to fully use the Macintosh environment and system services.

PCCard 3.0 is designed to handle single and multi-function cards. Support for well behaved cards is built into the system. There are options for adding support for ill-behaved cards at a minimum cost using card enablers, refer to “Card Enabler Interface” on page 32. Support for new technologies is implemented by means of plug-in Card enabler modules.

Support for Macintosh User experience customizing is provided through customizing card enabler plug-ins. Custom icons, card names and device names are all be available to the card manufacturer.

PCCard 3.0 has been designed to ensure that adequate testing can be performed. Automated test scripts will be created which will exercise all facets of the system. We believe using automated scripts in conjunction with the system testing by quality assurance engineers enables us to deliver a solid dependable product.

2.0 Related Documents

Designing PCI Cards and Drivers for Power Macintosh Computers, Apple Developer Press, 1995

Linux PCMCIA Programmers's Guide, Version 1.24, David Hinds, 7/31/95

PC Card Expansion for PowerBooks Computers, Apple Developer Press, 6/1/95

The PCMCIA Developer's Guide, Second Edition, Mori Welder, SYCARD Technologies, 1995

PCMCIA Primer, Larry Levine, M&T Books, 1995

PCMCIA System Architecture, Second Edition, Don Anderson, Mindshare, Inc., 1995

PC Card Standard, November 1995 Draft Printing

CardSoft[a] Technical Reference Rev.2.0 (SystemSoft, 4/95)

Inside Macintosh, Devices

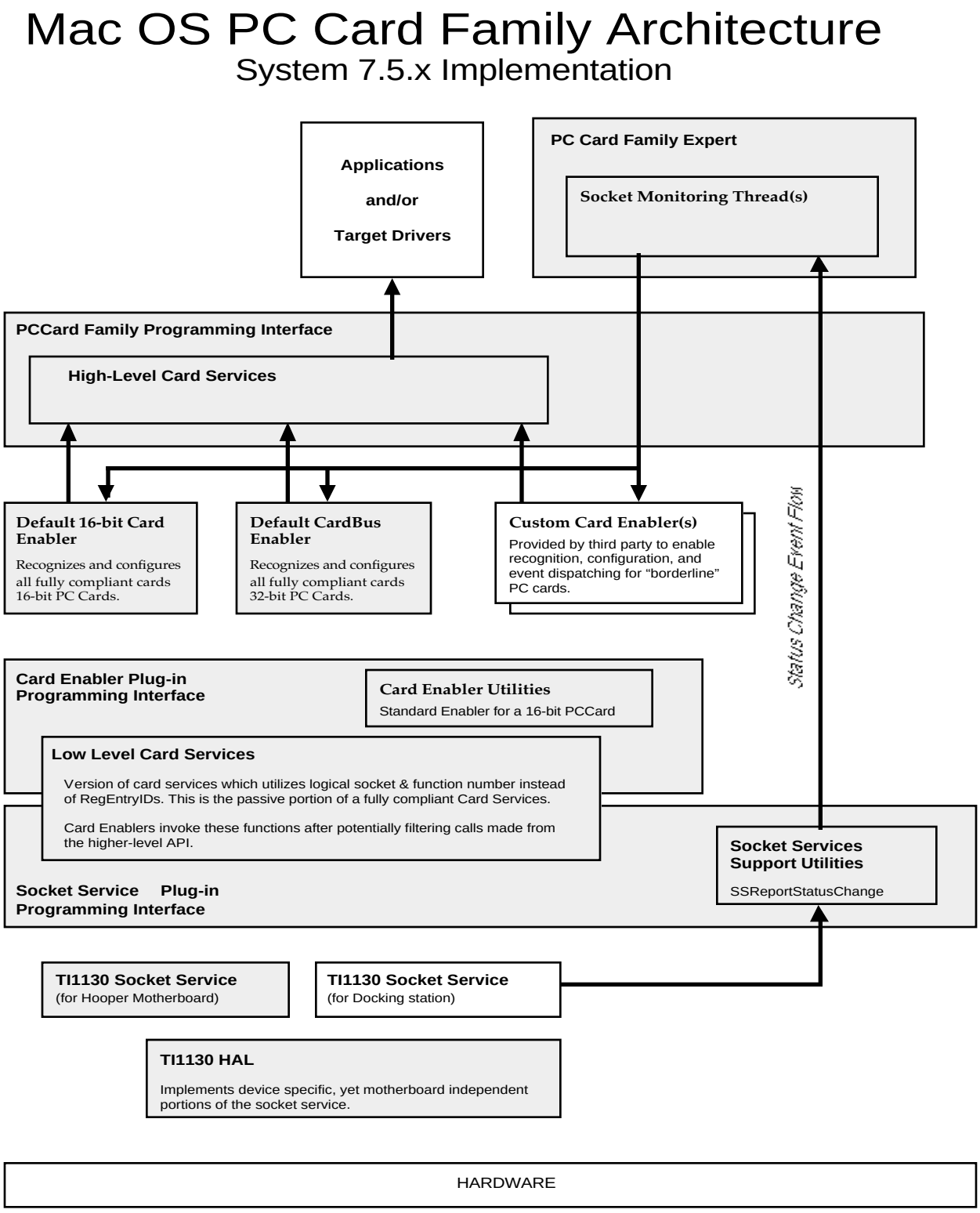
Inside Macintosh, Memory

Inside Macintosh, Inter-application Communication

3.0 About this Document

It is the intent of this document to describe the interfaces of the PC Card 3.0 Family. It is not the intent to cover all aspects of the Mac OS that relates to PC Card 3.0. Error codes returned by PC Card 3.0 described in this document are specfic to PC Card 3.0 and does not cover the return codes that may be returned by other parts of the Mac OS, example the name registry. Many routines in PC Card 3.0 will return the error code of the name registry or other parts of the Mac OS where it is appropriate.

FIGURE 1. PC Card Family Architecture, Event Processing



4.0 Architectural Elements

4.1 Applications and/or Target Driver

Devices which may be located on a PC Card will need a device driver. Ideally drivers will be bus-agnostic. For instance, the built in IDE driver should be able to service ATA cards without any PC Card specific modifications. Drivers are loaded by the Driver Loader, they typically receive information from the PC Card Family indirectly through the system Name Registry and Device Notification systems. In order to be “PC Card-aware” target drivers will have to handle power management and removability messages, but need not be aware of the source.

In the rare case where a driver needs to be aware that the device is located on a PC Card, we are providing the Card Services Family Programming Interface.

4.2 PC Card Family Expert

The PC Card Family Expert , supplied by SystemSoft, orchestrates the interaction of the other components described below – Card Enablers, Card Services and Socket Services.

During the boot process it supervises the initialization of the PC Card hardware and software. When there is a hardware interrupt from the PC Card socket, the Card Expert will assure that a PC Card is either properly inserted and readied for use or ejected. Finally it is responsible for passing on events generated by other parts of the operating system, such as Power Management events.

The PC Card Family Expert can be described as two tasks – a task that monitors the PC Card sockets and an administrative task that monitors PC Card software interaction with other parts of the system.

4.2.1 Socket Monitoring Task

The socket monitoring task listens for messages generated by the socket service plug-in in response to hardware interrupts. The socket monitoring task will load a card enabler (if necessary) and then call that card enabler to get things done.

During the boot process, the PC Card Expert will be loaded along with other family experts after the device drivers have been loaded and entered in the Name Registry. Internal Card Service, upon being loaded, will search for socket service plug-ins and initialize them. Internal Card Services will build a table that will keep track of sockets, RegEntries and allow virtualization of the sockets at the higher layers.

The PC Card Expert will register itself to be notified when hardware events occur on any of the sockets. Finally if a card is found to be inserted in a socket

when the machine boots, then the PC Card Expert will call the Card Enabler to make the card ready for use.

Once the system is up and running, the socket service plug-in will react to hardware interrupts generated by any of the PC Card sockets. When it has handled the interrupt, it will send a notification to the Card Expert. These notifications will be received when: 1) a card has been inserted, 2) a status change has occurred on a card, or 3) a card has been ejected. The Card Expert will call the appropriate routines in the Card Enabler to process these events.

4.2.2 Administrative Task

The administrative task is responsible for fielding messages from the Macintosh operating system. It will subscribe to services within the Mac OS that are relevant to the operation of the PC Card Family. These services include power management and device notification. Apple will have to supply several of these services since they do not already exist on System 7.5.2.

4.2.2.1 Power Management

It will be the responsibility of the administrative task to field messages from the power management service and notify the Card Enabler of a change in power states. These messages will include: 1) Battery Low, 2) Battery Dead, 3) System Sleep, and 4) System Shutdown.

4.3 Card Enabler

Enabler plug-ins are Code Fragments that have a well defined interface and a defined purpose. The Enabler plug-ins provide extensions to the PCCard family. Enablers are responsible for abstracting the intimate details of a card, its device(s) type, configuring the device(s), and placing this information into the name registry for target drivers to use when loading. The API between the enabler is generic enough to support current multi-device cards.

The enabler implements most of what would be the traditional Card Services client, less the driver code and state machines. SystemSoft will provide a generic enabler for standard multi-function cards.

SystemSoft will also provide the internals of the standard enabler as a shared library for writers of card-specific enablers. There will also be an easy way for developers to extend the standard enabler with static data (icons, card-specific settings, etc.).

4.4 Card Services Family Programming Interface

The Card Service Family Programming Interface(FPI) is the external card service interface used by target drivers (if necessary) and other traditional card service clients. The FPI is used to access and modify the configuration of PC Card devices. The interface supports the PC Card Card Service Spec-

ification. The MacOS binding designed by SystemSoft simplifies the card service API for developers and removes the historical IBM-PC functionality that is not necessary on the Macintosh.

The FPI mirrors the internal card service interface except that all FPI routines take a RegEntryRef as a parameter instead of a socket and device. Error reporting will take place if a device is not associated with a PC Card. Further, Device, Card and Socket RegEntry nodes may be interchanged wherever appropriate. The FPI is a thin layer between a client and the internal card service library. The Card Services FPI uses a “PCCard” prefix.

4.5 Card Enabler – the Generic Plug-In

Card enablers exist to implement standard card behavior and to provide a method of overloading standard card behavior with custom behavior. This is useful for developers of non-compliant or custom cards. The duties of the card enabler include card identification, card configuration, customizing of card services. Each card enabler provides a bottleneck for card services events.

4.6 Card Enabler Support Library

A collection of PC Card/Enabler utility routines (such as card identification routines) are available to enablers to make the process of identifying and processing cards easier to the developer.

4.7 Internal Card Service Library

All internal card services calls operate on a virtual socket and device id parameters. Card Enablers are implicitly registered with card services through the use of the PC Card expert and the name registry. All the traditional card service type functionality is handled by this library. Clients may explicitly register to receive card events if they wish.

5.0 Goals & Non-Goals

5.1 Short Term Goals

1. Support for Multi-function cards
The PC Card Family will support multi-function cards by design with single function cards treated as cards with one function.

2. Simplified API(s)
There are many places where the Card Services API, exposed by Opus, requires that client applications and drivers implement state-machines and handle interrupt-level callbacks to process simple asynchronous events.

The PC Card Family alleviates this chore by providing more natural shared library calls which provide the same functionality.

3. Support for CardBus will be integrated by providing a single 32-bit API for all functions[1].

4. Clear Separation of Driver and PCMCIA specific code
Device driver code concentrates on handling a specific device regardless of whether that device is located on the motherboard or a PC Card or a Card-Bus Card. This required drivers to be aware of power management and ejectability, but not specifically PCMCIA.

5. Minimize interrupt-level code
For performance as well as ease of programming and debugging, we want to minimize the amount of interrupt-level client code required.

5.2 Long Term Goals

1. Forward compatibility
Clients written to the PC Card FPI will be forward compatible with future Mac OS I/O model.

2. CardBus support
CardBus cards will be fully supported by the PC Card Family.

3. Bus agnostic Device Drivers
Ideally target device drivers can be written in a way that they do not have to be concerned how their device is physically connected to the computer. They need only retrieve information from the devices tree and be able to handle notifications regarding power management and ejection.

1. While CardBus will be explicitly supported in the design, the first implementation may not be required to handle CardBus cards due to lack of hardware.

5.3 Long Term Non-Goals

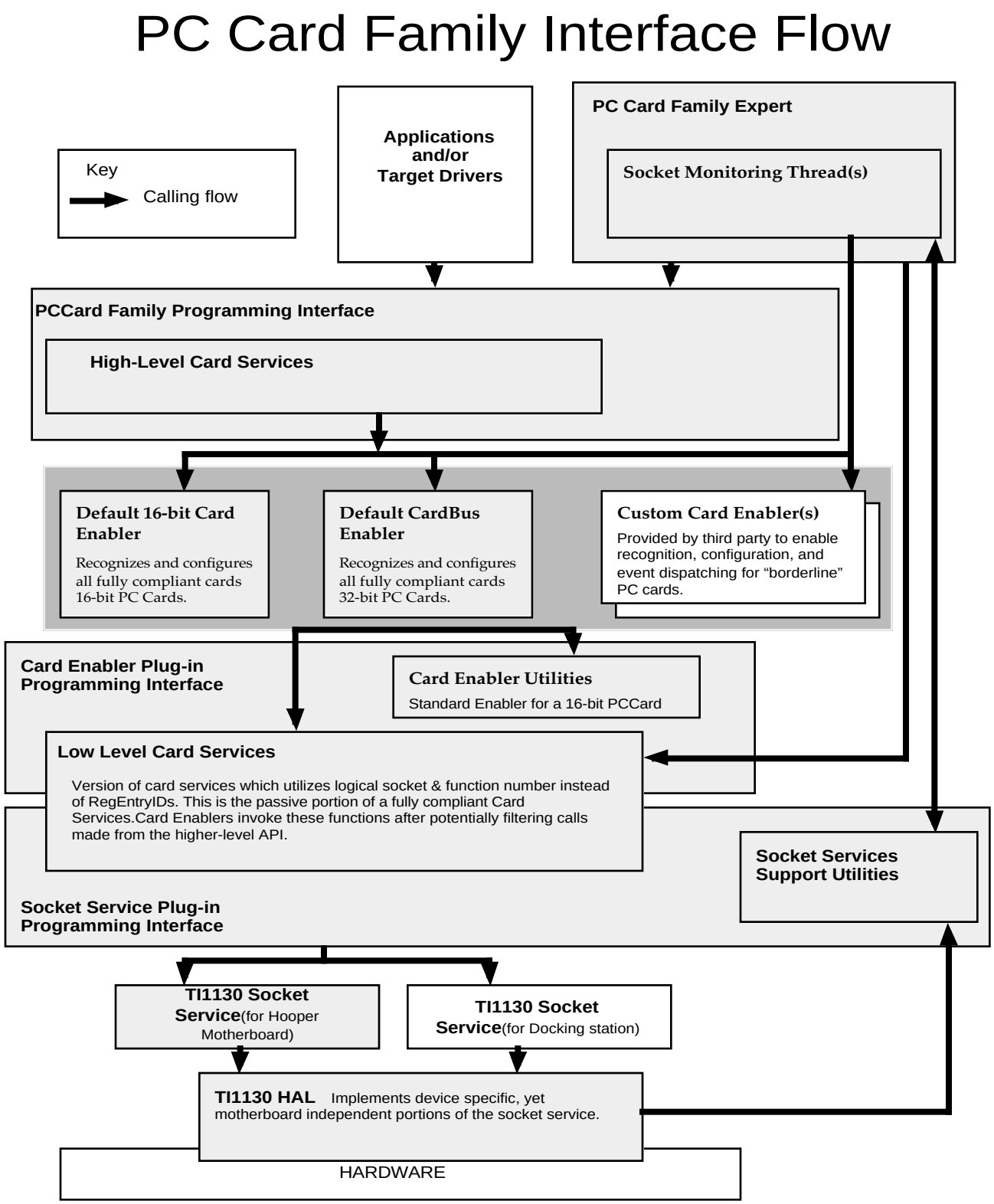
1. Compatibility with existing Card Services clients (System 7)
Since current client drivers cannot be supported on the target platform and most future hardware there is no need to provide support for current Card Services clients[1].

1. The reason for this is that, because the TREX controller guarantees a unique IO address space for each card, the RequestIO/ReleaseIO calls were omitted from the Opus API – unfortunately industry standard controllers do not allow us to make that assumption.

6.0 Terminology

- **Attribute Memory** - This address space contains the CIS and configuration registers for a card. Only the even bytes are implemented.
- **CardBus** - The CardBus cards have 32 bit data and address lines (shared). The CardBus uses the PCI bus protocol and supports bus mastering. The CardBus supports common, IO, configuration and expansion spaces.
- **Card Enabler** -the ‘low level’ expert, also known a card services client, this is a plug-in to the PcCard16 family expert.
- **Card Services** - Interface used by family expert and card enabler to configure the PC cards. It is also callable from target drivers through the FPI.
- **CIS** - Card Information Structure, the PcCard16 card configuration information. The CIS data is located on the card in attribute memory.
- **Common Memory** - The address space supports read and write memory access in 8 and 16 bit quantities.
- **Configuration Memory** - CardBus cards support special bus cycles to get to configuration registers and their CIS.
- **Mac OS 8** - The external code name of the successor version of the MacOS which will follow system version 7.5.x.
- **HBA** - Host bus adapter, i.e. the PC Card socket controller.
- **IO Memory**- This address space supports the IO access to a card.
- **PCMCIA** - Personal Computer Memory Card International Association
- **PC Card** - PcCard16 and CardBus cards, it is about the size of credit card
- **PcCard16** - The PcCard16 card supports data buses of 8 and 16 bits. It has an address space of 26 bits (64Mb) and supports common memory, attribute memory, and IO space. The bus protocol is ISA like
- **PcCard16 Family** - both the Card Services interface and family expert
- **PcCard16 Family Expert** - the high level expert it controls the HBA.
- **Socket controller** - the HBA that drives the PC Card sockets, controls memory and IO window mapping
- **Socket Services** -the PcCard16 family plug-in, it drives the HBA, it has a standard interface to card services.
- **Target Driver Plug-in** - The plug-in that drives a function on a PC Card.

FIGURE 2. PC Card Family Interface Calling Flow Diagram



7.0 External /Public Interfaces

The following sections cover in detail the public interfaces that are available to developers. The first section describes the PCCard Family Programming Interface (FPI). The second section is the Enabler support library interface which is available to developers who may need to develop a custom enabler to support non-compliant cards.

7.1 PCCard Family Programming Interface

The PCCard Family Programming Interface (PCCard FPI) is a thin PCCard Card Service binding layer that is exposed to the rest of the world. Target Drivers and other applications that wish to register with card service for event notification use the PCCard FPI to register and communicate with the PCCard family.

The Card Services interface as defined by PCMCIA forms the foundation of the PC Card Family Programming Interface. This standard interface has been adapted to Macintosh platform by adopting Mac OS I/O architectural elements wherever possible.

The Card Services programming interface can be divided into five main sections, as described in *PC Card Standard: Card Services Specfication:*

- Client Services
- Resource Management
- Client Utilities
- Bulk Memory Services
- Advanced Client Services

The major difference between this binding, and that which is provided in existing PCs, is that Card Services clients are now bound to the card, and not registered in a global list. Because drivers are instantiated on demand and replicated for each instance of a device, there is no need to keep a single global list of active clients.

Bulk Memory Services are NOT supported by Mac OS Card Services, as this functionality is better suited for use in a specialized Block Storage plug-in developed for linear flash arrays and/or other memory devices.

In addition to these change, several entry points have been removed and/or simplified to eliminate “DOS-isms” from the programming model.

The details of these differences are completely described in Section D.

7.1.1 Client Services

7.1.1.1 PCCardGetCardServicesInfo

The PCCardGetCardServicesInfo control call returns the current version information. The Card Services PC Card Standard compliance level for this specification is 0x0510.

OSStatus **PCCardGetCardServicesInfo** (
ItemCount * socketCount,
UInt32 * complianceLevel,
UInt32 * version);

Parameters:

← `socketCount`	- Current Number of Sockets
← `complianceLevel`	- Binary Coded Decimal value of the Card Services PCCard Standard Compliance Level
← `version`	- Binary Coded Decimal value of the Card Services implementation's version number

Return Codes:

noErr

7.1.1.2 PCCardRegisterClient

PCCardRegisterClient is provided to allow target drivers to register interest in PC Card status changes. When a PC Card status change occurs, the function provided in `clientCallback` is invoked.

NOTE: Unlike the x86/DOS binding, clients are registered for either a specific card in a socket or on all sockets in the system.

OSStatus **PCCardRegisterClient**(
const RegEntryRef* deviceID,
PCCardEventMask interestingEvents,
PCCardEventHandler clientCallback,
void * clientParam,
PCCardClientID * clientID)

Parameters:

→ `deviceID`	- Device identifier, enter a nil regentyref if you desire all sockets by creating a new regentryref.
→ `interestingEvents`	- Bit mask which events are interesting Table C.1
→ `clientCallback`	- Client supplied event handling function
→ `clientParam`	- Client supplied parameter passed to clientCallback
← `clientID`	- "ClientHandle"

Return Codes:

noErr	- If no error occured
kBadSocketErr	- if the socket is invalid

The clientCallback has the following format:

```
OSStatus (*PCCardEventHandler)(
                PCCardEvent     theEvent,
                PCCardSocket    vSocket,
                UInt32          device,
                UInt32          info,
                UInt32          MTDRequest,
                UInt32*         buffer,
                UInt32          misc,
                UInt32          status,
                void *          clientParam)
```

Parameters:

← `theEvent`	- The event that occurred, could be multiple events
← `vSocket`	- The virtual socket number of the card where the event occurred
← `device`	- The device number where the event occurred
← `info`	- information specific to the event being reported, refer to the PC Card Standard Card Service Specification for more details
← `MTDRequest`	- Specifically for MTD support(not supported by the PC Card 3.0)
← `buffer`	- Pointer to a buffer for modification by the client (not supported by the PC Card 3.0)
← `misc`	- argument used for miscellaneous information, refer to the PC Card Standard Card Service Specification for more details (Not used in Pc Card 3.0)
← `status`	- used by callback handlers to return information to Card Services
← `clientParam`	- Client parameter returned

A client event handler must perserve all callback entry arguments unless otherwise indicated. This ensures other callback handlers receive the same information and that Card Services may rely on the information when handlers have completed processing so it may perform any additional processing.

7.1.1.3 PCCardDeregisterClient

PCCardDeregisterClient is provided to unregister interest in PC Card status changes.

OSStatus **PCCardDeregisterClient**(
PCCardClientID clientID);

Parameters:

→ `clientID` - "ClientHandle"

Return Codes:

noErr	- If no error occured
kInvalidCSClientErr	- if the client Id is invalid

7.1.1.4 PCCardSetEventMask

PCCardSetEventMask allows the event mask to be changed

OSStatus **PCCardSetEventMask**(
PCCardClientID clientID,
PCCardEventMask interestingEvents)

Parameters:

→ `clientID`	-"ClientHandle"
→ `interestingEvents`	-Bit mask which events are interesting Table C.1

Return Codes:

noErr	- If no error occured
kInvalidCSClientErr	- if the client Id is invalid

7.1.1.5 PCCardGetEventMask

PCCardGetEventMask allows the client to check the event mask.

OSStatus **PCCardGetEventMask** (
PCCardClientID clientID,
PCCardEventMask * interestingEvents);

Parameters:

→ `clientID`	- "ClientHandle"
← `interestingEvents`	- Bit mask which events are interesting Table C.1

Return Codes:

noErr	- If no error occured
kInvalidCSClientErr	- if the client Id is invalid

7.1.1.6 PCCardRegisterTimer

The PCCardRegisterTimer call registers a callback structure with Card Services. Based on a tick count provided, Card Services calls the client back when the time period has elapsed and the Card Services interface is available. The client callback registered will be called when the timer elapses. A valid client handle must be obtained from calling PCCardRegisterClient.

OSStatus **PCCardRegisterTimer**(

PCCardClientID	registeredClientID,
PCCardTimerID	*lpNewTimerID,
long	delay)

Parameters:

→ `registerclientID`	- "ClientHandle"
← `lpNewTimerID`	- Timer ID
→ `delay`	- the number of ticks to wait, approximately 1 ms/tick.

Return Codes:

noErr	- If no error occured
kInvalidCSClientErr	- if the client Id is invalid
paramErr	- Bad parameter

7.1.1.7 PCCardDeRegisterTimer

PCCardDeRegisterTimer is provided to unregister timer clients.

OSStatus **PCCardDeRegisterTimer**(

PCCardTimerID	timerID);

Parameters:

→ `timerID`	- "timerID"

Return Codes:

noErr	- If no error occured
kNoClientTableErr	- The client table has not be initialized yet
kInvalidCSClientErr	- Card Services ClientID is not registered

7.1.1.8 PCCardGetStatus

The PCCardGetStatus control call returns the current status of a PC Card of the specified DeviceID.

OSStatus **PCCardGetStatus** (

const RegEntryRef*	deviceID,
UInt32 *	currentState,
UInt32 *	changedState,
PCCardVoltage*	Vcc,
PCCardVoltage*	Vpp);

Parameters:

→ `deviceID`	- Device identifier
← `currentState`	- current state of the socket
← `changedState`	- delta between the last time the socket service getstatus call was made, note that this will not be from the last time a particular client called Table C.2
← `Vcc`	- Vcc power applied to the socket
← `Vpp`	- Vpp power applied to the socket

Return Codes:

noErr	- if socket and function numbers are valid
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoCardErr	- if a card is not present
paramErr	- if a parameter is incorrect

7.1.2 Resource Management

7.1.2.1 PCCardRequestWindow

The PCCardRequestWindow control call allocates a range of system address space to a PC Card of the specified DeviceID.

When an IO address range is requested, the IO range is only allocated and reserved. The PCCardRequestConfiguration control call must be invoked to enable access to the IO range.

OSStatus **PCCardRequestWindow** (

const RegEntryRef*	deviceID,
PCCardWindowAttributes	windowAttributes,
LogicalAddress *	windowBase,
ByteCount*	windowSize,
PCCardAccessSpeed*	windowSpeed,
PCCardWindowOffset*	windowOffset
PCCardWindowID *	windowID);

Parameters:

→ `deviceID`	- Device identifier
→ `windowAttributes`	-window attributes Table C.3
↔ `windowBase`	-Window base address in bytes
↔ `windowSize`	-Minimum window size in bytes (Used as input for 16-bit PC Cards only)
↔ `windowSpeed`	- Window speed (16-bit PC Cards only)
← `windowID`	- Window Identifier

Return Codes:

noErr	- if all parameters are valid and request can be serviced
paramErr	- if a parameter is incorrect
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kBadSizeErr	- if the requested window size cannot be accommodated
kBadSpeedErr	- if the requested access speed is invalid or cannot be accommodated
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated
kOutOfResourceErr	- if no system address range is available to accommodate the request
kNoCardErr	- if no PC Card is present in the socket

7.1.2.2 PCCardReleaseWindow

The PCCardReleaseWindow control call disables and deallocates the system address space previously assigned to a PC Card by the PCCardRequestWindow control call.

OSStatus **PCCardReleaseWindow** (
PCCardWindowID windowID);

Parameters:

→ `windowID`	- Window Identifier

Return Codes:

noErr	- if window handle is valid
kBadHandleErr	- if window handle is invalid

7.1.2.3 PCCardModifyWindow [16-bit PC Card Memory Only]

The PCCardModifyWindow control call allows the Access Speed and/or Card Offset of a 16-bit PC Card memory window to be modified.

OSStatus **PCCardModifyWindow** (
PCCardWindowID windowID,
PCCardWindowAttributes windowAttributes,
PCCardAccessSpeed windowSpeed,

PCCardWindowOffset windowOffset);

Parameters:

→ `windowID`	-Window Identifier
→`windowAttributes`	- Window attributes Table C.3
→`windowSpeed`	- Window speed
→`windowOffset`	- PC Card memory offset

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadHandleErr	- if window handle is invalid
kBadAttributeErr	- if any attributes are invalid
kBadSpeedErr	- if the requested access speed is invalid or cannot be accommodated
kBadOffsetErr	- if the card offset is invalid
kNoCardErr	- if no PC Card is present in the socket

7.1.2.4 PCCardRequestConfiguration

The RequestConfiguration control call configures the PC Card of the specified DeviceID.

All IO windows previously assigned by PCCardRequestWindow are enabled for access.

OSStatus **PCCardRequestConfiguration** (

const RegEntryRef*	deviceID,
PCCardConfigOptions	configOptions,
PCCardInterfaceType	ifType,
PCCardCustomInterfaceID	ifCustomType,
PCCardVoltage	vcc,
PCCardVoltage	vpp,
LogicalAddress	configRegistersBase,
PCCardConfigPresentMask	configRegistersPresent,
PCCardFunctionConfigReg *	configRegisterValues);

Parameters:

→`deviceID`	- Device identifier
→`configOptions`	- configuration attributes Table C.4
→`ifType`	- Interface type Table C.5
→`ifCustomType`	- interface ID (for CustomIF)
→`vcc`	- Vcc voltage in tenths of volts
→`vpp`	- Vpp voltage in tenths of volts
→`configRegistersBase`	- 16-bit Card base address for registers
→`configRegistersPresent`	- 16-bit Card register values present bitmap
→`configRegisterValues`	- 16-bit Card register values byte array

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated
kBadTypeErr	- if the interface type is invalid
kBadVccErr	- if Vcc is invalid or unsupported
kBadVppErr	- if Vpp is invalid, unsupported or incompatible with Vcc
kBadArgsErr	- if the Custom ID is invalid
kNoCardErr	- if no PC Card is present in the socket

7.1.2.5 PCCardReleaseConfiguration

The PCCardReleaseConfiguration control call deconfigures the PC Card and socket.

All IO windows previously assigned by PCCardRequestWindow are disabled.

OSStatus **PCCardReleaseConfiguration** (
const RegEntryRef* deviceID);

Parameters:

→`deviceID`	- Device identifier

Return Codes:

noErr	- if the socket and function numbers are valid and there is a configuration to release
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device or no configuration to release

7.1.2.6 PCCardModifyConfiguration

The PCCardModifyConfiguration control call allows a PC Card configuration to be modified without having to issue PCCardRequestConfiguration and PCCardReleaseConfiguration calls.

OSStatus **PCCardModifyConfiguration**(
const RegEntryRef* deviceID,
PCCardConfigOptions configOptions,
PCCardVoltage vpp);

Parameters:

→`deviceID`	- Device identifier
→`configOptions`	- Changed Attributes Table C.4

→vpp - Vpp voltage in tenths of volts

Note: It is not valid to change anything but Vpp using PCCardModifyConfiguration.

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated
kBadVppErr	- if Vpp is invalid or unsupported
kNoCardErr	- if no PC Card is present in the socket

7.1.2.7 PCCardResetFunction

The PCCardResetFunction does a soft reset on the device specified.

OSStatus **PCCardResetFunction** (
const RegEntryRef* deviceID);

Parameters:

→deviceID	- Device identifier

Return Codes:

noErr	- valid deviceID and reset successful
kInvalidDeviceNumber	- invalid deviceID

7.1.3 Client Utilities

7.1.3.1 PCCardGetFirstTuple

The PCCardGetFirstTuple allows the tuples of the Card Information Structure to be read. To read any tuple, the desired tuple ID must be set to 0FFh. If no tuple data is required, the Data Buffer Length parameter must be set to zero. If the buffer supplied is not sufficient to handle the data to be returned the call will fill the buffer supplied with the tuple data without returning an error.

Note: If data is not returned then the tuple iterator is not advanced.

OSStatus **PCCardGetFirstTuple** (

const RegEntryRef*	deviceID,
PCCardTupleKind	desiredTuple,
PCCardTupleIterator	tupleIterator,
void *	dataBuffer,
UInt32*	dataBufferSize,
PCCardTupleKind *	foundTuple,

UInt32 * foundTupleSize);

Parameters:

→deviceID	- Device identifier
→desiredTuple	- FFh for any Tuple
↔tupleIterator	- Card Services Internal Use Only
→dataBuffer	- Pointer to Tuple data buffer
→dataBufferSize	- Length of Tuple data buffer in bytes
←foundTuple	- Tuple ID Found
←foundTupleSize	- Length of Tuple data found in CIS

Return Codes:

noErr	- if parameters are valid and request was serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoMoreItemsErr	- if specified tuple was not found
kNoCardErr	- if no PC Card is present in the socket

7.1.3.2 PCCardGetNextTuple

The PCCardGetNextTuple control call allows the tuples of the Card Information Structure to be read. To read any tuple, the desired tuple ID must be set to 0FFh. If no tuple data is required, the Data Buffer Length parameter must be set to zero. If the buffer supplied is not sufficient to handle the data to be returned the call will fill the buffer supplied with the tuple data without returning an error.

If an error occurs, the tuple interator IS NOT advanced.

OSStatus **PCCardGetNextTuple** (

const RegEntryRef*	deviceID,
PCCardTupleKind	desiredTuple,
PCCardTupleIterator	tupleIterator,
void *	dataBuffer,
UInt32*	dataBufferSize,
PCCardTupleKind *	foundTuple,
UInt32 *	foundTupleSize);

Parameters:

→deviceID	- Device identifier
→desiredTuple	- FFh for any Tuple
↔tupleIterator	- Card Services Internal Use Only Must be NULL for first Tuple ID read Must be preserved for subsequent Tuple ID reads
→dataBuffer	- Pointer to Tuple data buffer
→dataBufferSize	- Length of Tuple data buffer in bytes
←foundTuple	- Tuple ID Found
←foundTupleSize	- Length of Tuple data found in CIS

Return Codes:

noErr	- if parameters are valid and request was serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoMoreItemsErr	- if specified tuple was not found
kNoCardErr	- if no PC Card is present in the socket

7.1.4 AccessConfigurationRegister

AccessConfigurationRegister has been replaced by two calls to help reduce programmer error.

7.1.4.1 PCCardReadConfigurationRegister

The PCCardReadConfigurationRegister control call allows the PC Card configuration registers to be read.

```
OSStatus  PCCardReadConfigurationRegister (
              const RegEntryRef*          deviceID,
              PCCardConfigRegisterIndex   whichRegister,
              PCCardConfigRegisterOffset  offset,
              UInt8 *                     value);
```

Parameters:

→ `deviceID`	- Device identifier
→ `whichRegister`	- which register index to read
→ `offset`	- Memory Register Offset
← `value`	- Read Value

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kBadArgsErr	- if the register type or memory register offset is invalid
kNoCardErr	- if no PC Card is present in the socket

7.1.4.2 PCCardWriteConfigurationRegister

The WriteConfigurationRegister control call allows the PC Card configuration registers to be written.

```
OSStatus  PCCardWriteConfigurationRegister (
              const RegEntryRef*          deviceID,
              PCCardConfigRegIndex        whichRegister,
              PCCardConfigRegOffset       offset,
              UInt32                      value);
```

Parameters:

→ `deviceID`	- Device identifier

→ `whichRegister`	- Which register indes to modify
→ `offset`	- Memory Register Offset
→ `value`	- Read Value

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kBadArgsErr	- if the register type or memory register offset is invalid
kNoCardErr	- if no PC Card is present in the socket

7.1.5 Miscellaneous Interfaces

7.1.5.1 PCCardGetCardInfo

The PCCardGetCardInfo call returns information about the card such as type, sub-type, card name and vendor name.

```
OSStatus PCCardGetCardInfo(
            const RegEntryRef *  cardRef,
            PCCardDevType *      cardType,
            PCCardSubType *      cardSubType,
            StringPtr            cardName,
            StringPtr            vendorName)
```

Parameters:

→ `cardRef`	-Card identifier
← cardType	- type of card
← cardSubType	- subtype of card
← cardName	- Name fo the card
← vendorName	- name of the vendor

Return Codes:

noErr	- Success
paramErr	- if the card regentry ID is invalid
kInvalidRegEntryErr	- if no PC Card is present in the socket or the Card Identifier is invalid for the socket

7.1.5.2 PCCardEject

The PCCardEjectcontrol call physically ejects the PC Card of the specified DeviceID from the socket.

OSStatus **PCCardEject**(
const RegEntryRef* deviceID);

Parameters:

→ `deviceID`	-Device identifier

Return Codes:

noErr	- if ejection completed successfully
kInUseErr	- if card is still in use and eject request was denied
kNoCardErr	- if no PC Card is present in the socket

7.1.5.3 PCCardSetRingIndicate

The PCCardSetRingIndicate control call sets the ring indicate bit on and off on a modem card that supports it. [1]

OSStatus **PCCardSetRingIndicate**(
const RegEntryRef * deviceRef,
Boolean setRingIndicate)

Parameters:

→ `deviceRef`	-Device identifier
→ `setRingindicate`	-Boolean to turn the RingIndicate bit on and off on a modem card that supports Ring indicate

Return Codes:

noErr	- if operation completed successfully
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoCardErr	- if no PC Card is present in the socket
kUnsupportedModeErr	- If the card does not support Ring indicate

7.1.5.4 PCCardEnableModemSound

The PCCardEnableModemSound control call enables the client or driver to turn modem sound on or off.

OSStatus **PCCardEnableModemSound**(

1. **PCCardSetRingIndicate** this call may not be supported for all Mac OS platforms.

const RegEntryRef * cardRef,
Boolean enableSound)

Parameters:

→ `cardRef` -Device identifier
→ `enableSound` -Boolean to turn the sound off and on.

Return Codes:

noErr	- if operation completed successfully
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoCardErr	- if no PC Card is present in the socket
kUnsupportedModeErr	- If the card does not support sound

7.1.5.5 PCCardEnableZoomedVideoSound

The PCCardEnableZoomedVideoSound control call enables the client or driver to turn zoom video sound on or off.

OSStatus PCCardEnableZoomedVideoSound(
const RegEntryRef * cardRef,
Boolean enableSound)

Parameters:

→ `cardRef` -Device identifier
→ `enableSound` -Boolean to turn the sound off and on.

Return Codes:

noErr	- if operation completed successfully
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoCardErr	- if no PC Card is present in the socket
kUnsupportedModeErr	- If the card does not support sound

7.1.5.6 PCCardSetPowerLevel

The PCCardSetPowerLevel control call to set the power level of a device. PC Card 3.0 only knows and handles the kPCCardPowerOn and kPCCard-PowerOff state. If a developer wants to support low power state a custom enabler must be written to handle the kPCCardPowerLow state to handle the details of placing the card into low power.[1]

OSStatus PCCardSetPowerLevel(

1. PCCardSetPowerLevel call may not be supported for all platforms

const RegEntryRef * cardRef,
PCCardPowerOptions powerLevel)

Parameters:

→ `cardRef`	-Device or card identifier
→ powerLevel	-Power Level to place the device or card

Return Codes:

noErr	- if operation completed successfully
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
kNoCardErr	- if no PC Card is present in the socket
kUnsupportedModeErr	- If the card does not support sound

7.1.5.7 PCCardGetCardRefFromDeviceRef

The PCCardGetCardRefFromDeviceRef returns the RegEntryRef of the card given the device RegEntryRef. The card RegentryRef is the parent node of the device RegEntry created by the PC Card 3.0 software.

OSStatus **PCCardGetCardRefFromDeviceRef**(
const RegEntryRef * deviceRef,
RegEntryRef * cardRef)

Parameters:

→ `deviceRef`	- pointer to device RegEntryRef
← `cardRef`	- pointer to card RegEntryRef

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
paramErr	- if the register type or memory register offset is invalid
kNoCardErr	- if no PC Card is present in the socket
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

7.1.5.8 PCCardGetSocketAndDeviceFromDeviceRef

The PCCardGetSocketAndDeviceFromDeviceRef call returns the virtual socket number and device number given the RegEntrytRef of a device.

OSStatus **PCCardGetSocketAndDeviceFromDeviceRef**(

const RegEntryRef * deviceRef,
PCCardSocket * vSocket,
UInt32 * device)

Parameters:

→ `deviceRef`	- pointer to device RegEntryRef
← `vSocket`	- pointer to the virtual socket
← `device`	- pointer to the device number

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the device Identifier does not refer to a valid socket or device
paramErr	- if the register type or memory register offset is invalid
kNoCardErr	- if no PC Card is present in the socket
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

7.1.5.9 PCCardGetCardRef

The PCCardGetCardRef call returns the card RegEntryRef given a virtual socket number.

OSStatus **PCCardGetCardRef**(
PCCardSocket vSocket,
RegEntryRef * cardRef)

Parameters:

→ `vSocket`	- virtual socket number
← `cardRef`	- pointer to the card RegEntryRef

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
paramErr	- If the parameters car invalid
kNoCardErr	- if no PC Card is present in the socket
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

7.1.5.10 PCCardGetSocketRef

The PCCardGetSocketRef call returns the socket RegEntryRef given a virtual socket number.

OSStatus **PCCardGetSocketRef**(

```
                    PCCardSocket        vSocket,
                    RegEntryRef *       socketRef)
```

Parameters:

→ `vSocket`	- virtual socket number
← `socketRef`	- pointer to the socket RegEntryRef

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
paramErr	- If the parameters car invalid
kNoCardErr	- if no PC Card is present in the socket
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

8.0 Card Enabler Interface

8.1 Purpose

Card enablers exist to implement standard card behavior and to provide a method of overloading standard card behavior with custom behavior. This is useful for developers of non-compliant or custom cards. The duties of the card enabler include card identification, card configuration, and customization of card services. Each card enabler provides a bottleneck for card services events so that card specific event processing may be preformed by the enabler before the device driver is notified. It is the intention that a card vendor will not have to override any Card Enabler function but in certain cases were CIS information is not adequate on a card. A custom card enabler can correct information present in the card's the CIS information making the card appear to be compliant with the PC Card Standard.

It is important that the guidelines outlined in the following sections be followed exactly to ensure proper configuration of cards and of the Mac OS. Described in each call is the appropriate Card Enabler Support call to be used. They will be categorized as **mandatory**, **optional or "do not call" for custom enablers overriding the routine**.

8.2 Overview

Each Card Enabler Plug-in is an 'ndrv' (Native Driver). The ndrv must export a descriptor and a function table populated with custom entry points or Nils. The descriptor serves to identify the plug-in to facilitate matching of PC Cards with their card enablers. Card Enabler entry points are accessed through the Card Enabler Interface (_EI) layer. It is the responsibility of the Card Enabler to provide an interface for the PCCard FPI and Card Expert.

8.3 Plug-in File Type

All card enabler plug-ins must have a file type of 'ndrv'.

8.4 DriverDescriptor

The DriverDescriptor will be used to match a card enabler with a PC Card.

```
DriverDescription TheDriverDescription = {
        /*
         * Signature info
         */
        kTheDescriptionSignature,    /* OSType driverDescSignature*/
        kInitialDriverDescriptor,    /* DriverDescVersion driverDescVersion*/
        /*
         * DriverType driverType - these are defined in
         */
        kCompatiblePluginName,                      /* Name of hardware */
        kVersionMajor, kVersionMinor,               /* NumVersion version */
        kVersionStage, kVersionNonRel,

        /*
         * DriverOSRuntime driverOSRuntimeInfo
         */
        0
        /* RuntimeOptions driverRuntime*/
        | (0 * kDriverIsLoadedUponDiscovery)/* Loader runtime options*/
        | (0 * kDriverIsOpenedUponLoad)     /* Opened when loaded               */
        | (1 * kDriverIsUnderExpertControl) /* I/O expert handles loads/opens*/
        | (0 * kDriverIsConcurrent)         /* Not concurrent yet               */
        | (0 * kDriverQueuesIOPB),          /* Not internally queued yet*/
        kCompatiblePluginName,              /* Str31 driverName(OpenDriver param)*/
        0, 0, 0, 0, 0, 0, 0, 0,             /* UInt32 driverDescReserved[8]*/

        /*
         * DriverOSService Information. This section contains a vector count
         * followed by
         * a vector of structures, each defining a driver service.
         */
        1,                                  /*     ServiceCount nServices      */

        /*
         * DriverServiceInfo service[0]
         */
        kPCCardServiceCategory,             /* OSType serviceCategory    */
        kPCCardCardEnablerServiceType,      /* OSType serviceType        */

        1, 0,developStage, 1                /* version of the Open Transport    */
                                            /* programming interface that this  */
                                            /* driver supports                  */
                                            /* should be kOTDriverAPIVersion    */
};
```

8.5 Card Enabler loading

The PC Card 3.0 Family utilizes the Driver Loader library to load the appropriate card enabler for a card inserted in the system. Using the utilities supplied by the Driver Loader Library the PC Card 3.0 Family scans the

extensions folder for all appropriate drivers with service types ‘ndrv’. The best candidate is based on the card node name created by the PC Card 3.0 Family.

8.6 Card Enabler Plug-in Entry Points.

Entry points to the card enabler plug-in have been defined. These entry points provide the services required to match plug-ins with cards, get card information, prepare a card, and handle card events.

Plug-ins are required to fill in a function table. This function table is used to perform all actions required to set up a card. The current structure of this function table is as follows:

8.6.1 Card Enabler Plug-in typedefs

```
enum {
       kServiceTypePCCardEnabler='card',
       kPCCardEnablerPluginVersion= 0x00000001,
       kPCCardEnablerPluginCurrentVersion = kPCCardEnablerPluginVersion
};

/*     Card Enabler Entrypoints*/

typedef OSStatus (*CEValidateHardwareProc)(const RegEntryRef *cardRef);

typedef OSStatus (*CEInitializeProc)(const RegEntryRef *cardRef,
                              Boolean replacingOld);

typedef OSStatus (*CECleanupProc)(const RegEntryRef *cardRef,
                              Boolean beingReplaced);

typedef OSStatus (*CEPowerManagementProc)(const RegEntryRef *lpCardEntry,
                              PCCardPowerOptions powerLevel);

typedef OSStatus (*CEHandleEventProc)(const RegEntryRef *cardRef,
                              PCCardEvent theEvent);

typedef OSStatus (*CEGetCardInfoProc)(const RegEntryRef *cardRef,
                              PCCardDevType *cardType, PCCardSubType *cardSubType,
                              StringPtr cardName, StringPtr vendorName);

typedef OSStatus (*CEAddCardPropertiesProc)(const RegEntryRef *cardRef);

typedef OSStatus (*CEGetDeviceCountProc)(const RegEntryRef *cardRef,
                              ItemCount *numberOfDevices);

typedef OSStatus (*CEGetDeviceNameProc)(UInt32 socketNumber, UInt32 deviceNumber,
                              char *deviceName);

typedef OSStatus (*CEGetDeviceCompatibleProc)(const RegEntryRef *deviceRef,
                              UInt32 socketNumber, UInt32 deviceNumber, char *name);
```

```
typedef OSStatus (*CEGetDeviceTypeProc)(const RegEntryRef *deviceRef,
                              UInt32 socketNumber, UInt32 deviceNumber,
                              PCDeviceType *lpDeviceType);

typedef OSStatus (*CEGetDeviceTypeNameProc)(const RegEntryRef *deviceRef,
                              UInt32 socketNumber, UInt32 deviceNumber, char *name);

typedef OSStatus (*CEAddDevicePropertiesProc)(const RegEntryRef *deviceRef,
                              UInt32 device);

typedef OSStatus (*CEConfigureDeviceProc)(const RegEntryRef *deviceRef,
                              UInt32 deviceNumber);

typedef OSStatus (*CEFinalizeDeviceProc)(UInt32 socket, UInt32 device,
                              const RegEntryRef *deviceRef);

typedef OSStatus (*CEValidateCISProc)(UInt32 socket, UInt32 device,
                              UInt32 *lpCISChainCount);

typedef OSStatus (*CEGetFirstTupleProc)(UInt32 socket, UInt32 device,
                              PCCardTupleIteratorPtr lpTupleIterator,
                              Byte desiredTuple, void *lptupleData,
                              UInt32 *lpTupleBufferSize, Byte *lpFoundTuple);

typedef OSStatus (*CEGetNextTupleProc)(PCCardTupleIteratorPtr lpTupleIterator,
                              Byte desiredTuple, void *lptupleData,
                              UInt32 *lpTupleBufferSize, Byte *lpFoundTuple);
```

8.6.2 Card Enabler Dispatch Table structure

```
struct PCCardEnablerPluginHeader {
       UInt32               pluginDispatchTableVersion;
       UInt32               Reserved1;
       UInt32               reserved2;
       UInt32               reserved3;
};
typedef struct PCCardEnablerPluginHeader PCCardEnablerPluginHeader;

struct PCCardEnablerPluginDispatchTable {
       PCCardEnablerPluginHeader header;

       /* General functions*/
       CEValidateHardwareProc       validateHardwareProc;
       CEInitializeProc             initializeProc;
       CECleanupProc                cleanUpProc;
       CEPowerManagementProc        setPCCardPowerLevel;

       /* Card functions*/
       CEHandleEventProc            handleEventProc;
       CEGetCardInfoProc            getCardInfoProc;
       CEAddCardPropertiesProc      addCardProperties;
       CEGetDeviceCountProc         getDeviceCount;

       /* Device functions*/
       CEGetDeviceNameProc          getDeviceName;
       CEGetDeviceCompatibleProc    getDeviceCompatibleNames;
       CEGetDeviceTypeProc          getDeviceType;
```

```
        CEGetDeviceTypeNameProc      getDeviceTypeName;
        CEAddDevicePropertiesProc    addDeviceProperties;
        CEConfigureDeviceProc        configureDevice;
        CEFinalizeDeviceProc         finalizeDevice;

        /* Card Services Overrides...*/
        CEValidateCISProc            validateCIS;
        CEGetFirstTupleProc          getFirstTuple;
        CEGetNextTupleProc           getNextTuple;

        /* InterruptHandlers...*/
        InterruptHandler             cardInterruptHandlerFunction;
        InterruptEnabler             cardInterruptEnableFunction;
        InterruptDisabler            cardInterruptDisableFunction;
};

typedef struct PCCardEnablerPluginDispatchTable PCCardEnablerPluginDispatchTable;

typedef PCCardEnablerPluginDispatchTable *PCCardEnablerPluginDispatchTablePtr;
```

8.6.3 initializeProc

This routine is called to populate the name registry for all devices on a card. The table of function pointers which is handed in, contains the pointers to custom routines supplied by the enabler or default routines supplied by the CardEnablerLib, a shared library supplied by SystemSoft.

Mandatory that CEInitializeCard is called before performing custom enabler specific code.

OSStatus **initializeProc**(const RegEntryRef *cardRef,
Boolean replacingOld);

Parameters:

→ `cardRef`	- Device identifier
→ `replacingOld`	-Boolean stating if the call is being called during the replacement process.

Return Codes:

noErr	- Success
paramErr	- Bad parameter
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

8.6.3.1 Example Code of custom enabler table

```
PCCardEnablerPluginDispatchTable ThePluginDispatchTable =
        {
        /* PCCardEnablerPluginHeader        */
```

```
{kPCCardEnablerPluginCurrentVersion, 0, 0, 0},

/* CEValidateHardwareProc           */     MyValidateHardwareProc,
/* CEInitializeProc                 */     CEInitializeCard,
/* CECleanupProc                    */     CEFinalizeCard,
/* CEPowerManagementProc            */     CEPowerManagement,

/* CEHandleEventProc                */     CEHandleCardEvent,
/* CEGetCardInfoProc                */     CEGetCardInfo,
/* CEAddCardPropertiesProc          */     CEAddCardProperties,
/* CEGetDeviceCountProc             */     CEGetDeviceCount,

/* CEGetDeviceNameProc              */     MyGetDeviceName,
/* CEGetDeviceCompatibleProc                */
                                            CEGetDeviceCompatibleNames,
/* CEGetDeviceTypeProc                      */     MyGetDeviceType,
/* CEGetDeviceTypeNameProc                  */     MyGetDeviceTypeName,
/* CEAddDevicePropertiesProc                */     CEAddDeviceProperties,
/* CEConfigureDeviceProc                    */     CEConfigureDevice,
/* CEFinalizeDeviceProc                     */     CEFinalizeDevice,

/* CEValidateCISProc                        */     CEValidateCIS,
/* CEGetFirstTupleProc                      */     CSGetFirstTuple,
/* CEGetNextTupleProc                       */     CSGetNextTuple,

/* InterruptHandler                         */
                                            CEDefaultInterruptHandler,
/* InterruptEnabler                         */     NULL,
/* InterruptDisabler                        */     NULL
};
```

8.6.4 cleanUpProc

This entry point is called immediately before the plug-in is unloaded and allows the enabler developer to perform any necessary clean up before the plug in is removed. This call is only made if the card enabler plug-in was successfully matched with a card.

Mandatory that CEFinalizeCard is called after the enabler cleans up.

OSStatus **cleanUpProc** (
const RegEntryRef *cardRef,
Boolean beingReplaced);

Parameters:

→ `cardRef`	- Device identifier
→ `replacingOld`	-Boolean stating if the call is being called during the replacement process.

Return Codes:

noErr	- Success

paramErr	- Bad parameter
kInvalidRegEntryErr	- The RegEntryRef for the device is invalid

8.6.5 validateHardwareProc

Determine whether the card in question is supported by the enabler.

Every custom enabler must have a **validateHardwareProc** the Card Enabler support library does not supply a default CEValidateHardware call.

OSStatus **validateHardwareProc**(
const RegEntryRef *cardRef)

Parameters:

→ `cardRef`	- Device identifier

Return Codes:

noErr	- Success, the card will be handled by the enabler
kNotMyCardErr	- The card will not be handled by the enabler

8.6.6 getFirstTuple

This function allows enabler writers to override the default card services function “CSGetFirstTuple.”

CEGetFirstTuple is optional, it is suggested that this call be used only to fix known tuple processing problems such as missing tuples.

OSStatus **getFirstTuple** (

PCCardSocket	socket,
UInt32	device,
PCCardTupleIteratorPtr	lpTupleIterator,
Byte	desiredTuple,
void *	lptupleData,
UInt32 *	lpTupleBufferSize,
Byte *	lpFoundTuple)

Parameters:

→ `socket`	- socket number
→ `device`	- device number
→ `lpTupleIterator`	- A tuple iterator for card services to use
→ `desiredTuple`	- The tuple code for the desired tuple
↔ `lptupleData`	- A pointer to a buffer to fill or nil
↔ `lpTupleBufferSize`	- A pointer to a UInt32 to receive the data size
← `lpFoundTuple`	- A pointer to receive the kind of tuple returned.

Return Codes:

noErr	- Success

8.6.7 getNextTuple

This function allows enabler writers to override the default card services function “CSGetNextTuple.”

CSGetNextTuple is optional, it is suggested that this call be used only to fix known tuple processing problems such as missing tuples.

```
OSStatus getNextTuple (
        PCCardTupleIteratorPtr      lpTupleIterator,
        Byte                        desiredTuple,
        void *                      lptupleData,
        UInt32 *                    lpTupleBufferSize,
        Byte *                      lpFoundTuple)
```

Parameters:

→ `lpTupleIterator`	- A tuple iterator for card services to use
→ `desiredTuple`	- The tuple code for the desired tuple
↔ `lptupleData`	- A pointer to a buffer to fill or nil
↔ `lpTupleBufferSize`	- A pointer to a UInt32 to receive the data size
← `lpFoundTuple`	- A pointer to receive the kind of tuple returned.

Return Codes:

noErr	- Success

8.6.8 handleEventProc

This entry point is called in response to card events. This entry point will only be called in response to card events identified by the enabler descriptor's EventMask field.

It is mandatory to call **CEHandleEvent unless the custom enabler wants to stop the event from being processed.**

```
OSStatus handleEventProc(
        const RegEntryRef*          deviceID,
        PCCardEvent                 theEvent)
```

handleEventProc is responsible for delivering events to the device driver(s) loaded for it’s particular card. This allows the normalization of events for misbehaved cards by it’s enabler.

Parameters:

→ `deviceID`	- Device identifier
← `theEvent`	- Card event from Socket Services Table C.1

Return Codes:

noErr	- Success

8.6.9 AddCardPropertiesProc

This is the main entry point used to populate the name registry for all devices on a card. The name registry must be completely populated if this entry point returns successfully.

There is no mandatory call for this routine.

```
OSStatus CEAddCardPropertiesProc(
            const RegEntryRef *   cardRef)
```

Parameters:

→ `cardRef`	- Card identifier

Return Codes:

noErr	- Success

8.6.10 AddDevicePropertiesProc

This function must completely populate the name registry for the specified device on a card.The name registry must be completely populated for the device if this entry point returns successfully.

There is no mandatory call for this routine.

```
OSStatus AddDevicePropertiesProc(
                  const RegEntryRef *  deviceRef,
                  UInt32               device);
```

Parameters:

→ `deviceRef`	- device identifier
→ `deviceNumber`	- index of device to set up

Return Codes:

noErr	- Success

8.6.11 getDeviceCount

This function returns the number of devices on a card.

It is optional that **CEGetDeviceCount** be called.

OSStatus **getDeviceCount**(
const RegEntryRef* cardID,
UInt32 * pDeviceCount)

Parameters:

→ `cardID` - Card identifier
← `pDeviceCount` - Number of devices on card

Return Codes:

noErr - Success

8.6.12 getDeviceType

This function returns the type of a device on a card.

It is optional that **CEGetDeviceType be** called.

OSStatus **getDeviceType**(
const RegEntryRef* deviceRef,
PCCardSocket socket,
UInt32 device,
PCDeviceType pDeviceType);

Parameters:

→ `deviceRef` - device identifier
→ `socket` - socket number
→ `device` - device number
← `pDeviceType` - type of device
Table C.6

Return Codes:

noErr - Success

8.6.13 getDeviceTypeName

This function returns the name of the type of device on a card.

It is optional that **CEGetDeviceTypeName** be called.

OSStatus **getDeviceTypeName**(

const RegEntryRef *	deviceRef,
PCCardSocket	vSocket,
UInt32	device,
char *	name;

Parameters:

→ `deviceRef`	- device identifier
→ `vSocket`	- socket number
→ `device`	- device number
← `name`	- name of device type

Return Codes:

noErr	- Success

8.6.14 getDeviceName

This function returns the name of the device on a card.

It is optional that **CEGetDeviceName** be called.

OSStatus **getDeviceName**(

PCCardSocket	vSocket,
UInt32	device,
char *	name);

Parameters:

→ `vSocket`	- socket number
→ `device`	- device number
← `name`	- name of device type

Return Codes:

noErr	- Success

8.6.15 getCardInfoProc

This functions returns information about the card to be used by the finder extension for display of the cards icon and information of the card.

It is optional that the CEGetCardInfo be called.

OSStatus CEGetCardInfo(

const RegEntryRef *	cardRef,
PCCardDevType *	cardType,

PCCardSubType * cardSubType,
StringPtr cardName,
StringPtr vendorName)

Parameters:

→ `cardRef`	- card identifier
← cardType	-type of card
← `cardSubType`	- sub type of card
← card`Name`	- name of card
← vendor`Name`	- name of vendor

Return Codes:

noErr	- Success
paramErr	- parameter error

8.6.16 addDeviceProperties

This function is required to configure the specified device on the card and populate the name registry with memory window information in the assigned-address and APPL,address fields.

There is no mandatory call for this routine.

OSStatus **addDeviceProperties**(
const RegEntryRef * deviceRef,
UInt32 device);

Parameters:

→ `deviceID`	- Device identifier
→device	- device number

Return Codes:

noErr	- Success

8.6.17 cardInterruptHandlerFunction

This function is of type InterruptHandler and is responsible for processing card interrupts as defined in Interrupts.h.

8.6.18 cardInterruptEnableFunction

This function is of type InterruptEnabler and is responsible for processing card interrupts as defined in Interrupts.h.

8.6.19 cardInterruptDisableFunction

This function is of type InterruptDisabler and is responsible for processing card interrupts as defined in Interrupts.h.

8.7 Card Enabler Usage by the PC Card 3.0 Family

This section will explain the usage and control flow of a custom card enabler in detail. There are four stages in the life of an enabler, card insertion, card ejection, replacement and event notification. In each of these stages different routines are called at specific times to allow an enabler to perform defined tasks. The following information is supplied as a guide to assist the developer in designing custom card enablers. It is important to note that the PC Card family will read tuples using **getFirstTuple** only, it will use the **getNextTuple** call only when looking for multiple CISTPL_CFTABLE_ENTRYs entries.

8.7.1 Card Insertion Processing

When a card is inserted the PC Card 3.0 family is notified and starts to process the card. The PC Card Family will create a card entry in the name registry for the card inserted.

First the expert will call CSValidateCIS - this will insure that your enabler can read the CIS on the inserted card. (Ironically, a **kBadCISErr** is not considered a fatal error, so we will ignore invalid checksums - However if the card does not come ready, or some other error prevents us from reading all of the CIS tuples, then the card recognition process will be terminated.)

Next the expert will seek to generate a card name which will be used for the Name Registry Entry. The expert will look for a CISTPL_MANFID tuple and will transform it into a name using an algorithm that is described below. If there is no MANFID, the expert seeks to generate a name from the manufacturer name in the CISTPL_VERS_1 tuple. If there is no VERS1 tuple either, the name will be "pccard". You need not be concerned with the algorithm since then name will appear in the name registry in two places. If the generated name is "pccard12,403", the Card registry entry will be named "Devices:device-tree:bandit:ti1130:pccard12,403" and it will have a "name" property which will have a value of "pccard12,403". The Card registry entry will also have a "compatible" property which will be used later on to find an enabler.

After the Card registry entry has been created, the expert will look for card enablers that might want to handle this card. The enablers are opened in

order and the first enabler that indicates it can handle the card will be given control and the search will be ended.

First the expert looks for an enabler that matches the "name" property in the Card RegEntry. The name must exactly match the driverName field of the DriverOSRuntime portion of the DriverDescription structure that is exported by your enabler. (See DriverFamilyMatching.h for the details, or look at the custom enabler sample in the SDK.) If a driver with an exact name match is found, then the driver is loaded and the **validateHardwareProc** in the **PCCardEnablerPluginDispatchTable** is called. Your enabler should examine the card's CIS to assure that you want to handle it. If you do want to handle the card, then return **noErr** from your validateHardwareProc. If you do not want to handle it, return an error code - **kUnsupportedCardErr** would be a good choice. If your enabler returns **noErr**, then it will be kept loaded and the search for enablers will terminate. If your enabler returns an error, it will be unloaded and the search continues.

If there is no exact name match card enabler or it refuses to handle the card, we continue searching using the "compatible" property of the Card RegEntry. The compatible property has two zero terminated strings in it.

If no Custom Enabler has accepted the card (or none exist), then the search will proceed to the "compatible" property. The "DefaultPCCardEnabler" is part of the system software and it's **validateHardwareProc** will always return noErr.

Once an enabler has been selected, the expert continues the card recognition and configuration process. The **initializeProc** of selected enabler is called. A Device Registry entry (or entries) will be created and the devices configured. All access to the card will be done through the **PCCardEnablerPluginDispatchTable** of the card enabler - so your enabler can override the CIS information that exists on the card.

Once the expert has finished the configuration process, it will load target drivers for each of the devices that were found on the card. It will use a driver matching algorithm that first looks at the "name" property of the device RegEntry and the "compatible" property.

The devices on the card are now ready to use by the Finder and/or applications.

notes:

1) if an error code is returned to the expert, an "AAPL,pccard-error" property will be added to the Card RegEntry. The value will be the first error code returned to the expert.

2) here is an example of the algorithm used to generate a card name for the registry

Suppose a card has a MANFID of 12000304

- separate fields 1200 and 0304

- byte swap them, yielding 0012 and 0403

- suppress leading zeros, yielding 12 and 403

- build a name string, "pccard12,403"

8.7.2 The Device Initialization

At this point there exists a card node RegEntry and the card enabler is loaded. The PC Card Family then starts to process the devices on the card. The family determines the number of devices on the card by calling **getDeviceCount.**

The PC Card Family then gets information from the card to store in the card entry by calling **getFirstTupleProc** looking for the following tuples: CISTPL_VERS_1, CISTPL_FUNCID and CISTPL_FUNCE. The reason we need to do this is that many older cards “lose” their CIS once they are configured. A call is also made to **getDeviceTypeProc.** The information gathered during this process is stored in the card RegEntry using the name ‘PCCardInfo’. This data is used when the finder calls PCCardGetCardInfo and actually can be overridden by the **getCardInfoProc** enabler call.

At this point, the loaded enabler gets called using the **addCardProperties** calls**.** This allows the enabler to add any RegEntry fields to the card node. The default behavior at this point is to do nothing,

The PC Card Family is finished with the card node and starts to process the devices on the card. It first creates the device node calling **getDeviceName** to name the device and use the name in the name property. It determines the compatible field using the **getFirstTuple** looking for the CISTPL_FUNCID and CISTPL_FUNCE tuples, if they are not found then the card is assumed to be an SRAM card. The device type name is determined by calling getDeviceTypeName, the default call uses the CISTPL_VERS_1 and CISTPL_FUNCID tuple to determine the device type name.

The PC Card Family then calls **addDeviceProperties** so a custom enabler can add properties to the device node in the name registry. The card is not configured at this time because some cards may not allow access to the CIS once configured.

The next step is to configure the card. The PC Card Family calls **configureDevice** to start the configuration process. The **CEConfigureDevice** call in the Card Enabler Support Library should be called first in a custom enabler that overrides this call. **CEConfigureDevice** call looks for the CISTPL_CONFIG tuple to fill in a internal configuration table. It then gets all of the CISTPL_CFTABLE_ENTRYs using **getNextTuple.** It will parse, store and sort the CISTPL_CFTABLE_ENTRY entries in priority order. The PC Card Family will then attempt to configure the device/card using this information.

If the CISTPL_CONFIG tuple does not exist the device is considered unconfigurable. When a card is unconfigurable the support library call will attempt to

identify the device by calling **getDeviceType.** If the device type is memory it will attempt to determine the memory type and speed using the CISTPL_DEVICE tuple. If this fails the attempt to configure the card stops and an error code is returned.

8.7.3 Card Ejection

During a card ejection all clients for the particular device, socket or card are notified and have a chance to stop the card ejection if necessary, Note that if a card is paper clipped a client has no chance to stop it. The PC Card Family unloads the target driver for a card, powers the card down and ejects the card. It then will call **CleanUpProc** to give the enabler a chance to clean up the name registry before unloading it.

8.7.4 Event Notification

A custom card enabler may override the **handleEventProc**. The handleEventProc receives the PC Card events that will be sent to the clients registered for a particular socket, card, or device. This will allow a custom enabler to filter events or perform tasks that may be required before a client is notified. It is important that a custom enabler calls **CSNotifyClients** if it wants to ensure that registered clients get notified.

8.7.5 Enabler Replacement

The **finalizeDevice** enabler plug in call is used during replacement which will only occur with enablers that are in the system ROM.

8.8 Card Enabler Support Library

The Card Enabler Support Library is designed to support an enabler in card detection, assist in managing other clients of the card, and help maintaining the device information in the name registry and interrupt trees and to transparently provide default function table behavior.

8.8.1 Card Identification

One of the tasks delegated to the card enabler is card identification. This task typically involves examining the Card Information Structure (CIS). To assist the Card Enabler in this process, several Card Identification subroutines are provided in by the PC Card family.

8.8.1.1 CEGetCardType

This function provides a mechanism of identifying what type of PC Card is inserted in a PC Card slot (e.g., 16-bit memory card, 16-bit I/O card, Zoomed Video device, or CardBus card).

OSStatus **CEGetCardType**(
RegEntryRef cardEntry,
PCCardType* cardType);

Parameters

`cardEntry` - RegEntry ID of the card
`cardType` - pointer to card type
Table C.6

Return codes

noErr - compare succeeded

8.8.1.2 CECompareCISTPL_VERS_1

PCCompareCISTPL_VERS_1 compares the CISTPL_VERS_1 tuple with the specified parameters. All supplied parameters must match; you may pass nil to ignore a parameter.

OSStatus **CECompareCISTPL_VERS_1**(
RegEntryRef cardEntry,
Byte majorVersion,
Byte minorVersion,
const char* manufacturer,
const char* productName,
const char* info1,
const char* info2);

Parameters

`cardEntry` - RegEntry ID of the card
`majorVersion` - Major rev of card
`minorVersion` - Minor rev of card
`manufacturer` - manufacturer of card
`productName` - name of product
`info1` - extra info
`info2` - extra info

Return codes

noErr - compare succeeded
kPCCCompareFailedErr - compare failed

8.8.1.3 CECompareCISTPL_MANFID

PCCompareCISTPL_MANFID Compares the CISTPL_MANFID tuple with the specified parameters. All supplied parameters must match; you may pass nil to ignore a parameter.

```
OSStatus CECompareCISTPL_MANFID(
        RegEntryRef     deviceID,
        UInt16          manufacturerCode,
        UInt16          manufacturerInfo);
```

Parameters

`deviceID`	- PC Card's registry entry
`manufacturerCode`	- manufacturer code bytes
`manufacturerInfo`	- manufacturer info bytes

Return codes

noErr	- compare succeeded
kPCCompareFailedErr	- compare failed

8.8.1.4 CECompareMemory

PCCompareMemory compares a block of memory on a PC Card with a block of user supplied data In the case of attribute memory, offset must be even and every other byte is compared. This routine is intended to be used as a ‘last resort’ for PC Cards which cannot be identified using standard tuple processing techniques.

```
OSStatus CECompareMemory (
        RegEntryRef         deviceID,
        PCCardMemoryType    memType,
        ByteCount           offset,
        ByteCount           length,
        Byte *              dataToCompare);
```

Parameters

`deviceID`	- PC Card's registry entry
`memType`	- I/O, Attribute, memory space
offset	- Offset from beginning of address space
length	- Length of data to compare

dataToCompare - Data to compare

Return codes

noErr	- compare succeeded
kPCCompareFailedErr	- compare failed

8.9 Internal Card Services

8.9.1 Purpose

Internal Card Services supplies the lowest level programming interface to the card services library. The Internal Card Services programming interfaces require a logical socket number and a device number to identify a device. This differs from the Card Services Family Programming Interface which requires a RegEntryID to identify a device. These routines are either called directly or inderectly via a card enabler driver by the PC CardFamily Programming Interface. Client programs should not call the internal card service API directly.

8.9.2 Client Services

The client services interface consists of routines for client management and event notification. Clients consist of target drivers or applications that require event notification for a particular device or socket.

8.9.2.1 CSGetCardServicesInfo

The GetCardServicesInfo control call returns the current version information. The Card Services PC Card Standard compliance level for this specification is 0x0501, see page 59 of PC Card Socket Services specification, dated November 95.

OSStatus **CSGetCardServicesInfo** (

ItemCount *	socketCount,
UInt32 *	complianceLevel,
UInt32 *	version)

Parameters:

← `socketCount`	- Current Number of Sockets
← `complianceLevel`	- Binary Coded Decimal value of the Card Services PCCard Standard Compliance Level
← `version`	- Binary Coded Decimal value of the Card Services implementation's version number

Return Codes:

noErr	- Success

8.9.2.2 CSRegisterClient

CSRegisterClient is provided to allow target drivers to register interest in PC Card status changes. When a PC Card status change occurs, the function provided in `clientCallback` is invoked.

NOTE: Unlike the x86/DOS binding, clients are registered for events in a specific socket, rather than all status changes in the system. Passing kCSNotifyAllSockets as the socket of interest will register for all sockets.

OSStatus **CSRegisterClient** (

PCCardSocket	vSocket,
PCCardEventMask	interestingEvents,
PCCardEventHandler	clientCallback,
void *	clientParam,
PCCardClientID *	registeredClient)

Parameters:

→ `vSocket`	- which socket, or kCSNotifyAllSockets for all sockets.
→ `interestingEvents`	- Bit mask which events are interesting Table C.1
→ `clientCallback`	- Client supplied event handling function
→ `clientParam`	- Client supplied parameter passed to clientCallback
← `registeredClient`	- "ClientHandle"

Return Codes:

noErr	- if socket and function numbers are valid
kBadSocketErr	- if the socket is invalid
kBadDeviceErr	- if the device number is invalid

8.9.2.3 CSDeregisterClient

CSDeregisterClient is provided to unregister interest in PC Card status changes.

OSStatus **CSDeregisterClient**(PCCardClientID clientID)

Parameters:

→ `clientID`	- "ClientHandle"

Return Codes:

noErr	- if socket and function numbers are valid
kBadClientIDErr	- if the client ID is invalid

8.9.2.4 CSSetEventMask

SetEventMask allows the event mask to be changed

OSStatus **CSSetEventMask**(

PCCardClientID	clientID,
PCCardEventMask	interestingEvents)

Parameters:

→ `clientID`	-"ClientHandle"

→ `interestingEvents` -Bit mask which events are interesting
Table C.1

Return Codes:

noErr	- Success
kBadClientIDErr	- if the client ID is invalid

8.9.2.5 CSGetEventMask

GetEventMask allows the client to check the event mask.

OSStatus **CSGetEventMask** (
PCCardClientID clientID,
PCCardEventMask * interestingEvents)

Parameters:

→ `clientID`	- "ClientHandle"
← `interestingEvents`	- Bit mask which events are interesting Table C.1

Return Codes:

noErr	-Success
kBadClientIDErr	- if the client ID is invalid

8.9.2.6 CSRegisterTimer

CSRegisterTimer is provided to allow target drivers to register a timer with card services to be notified via the callback registered in the registerClient call. A valid client handle must be first obtained from calling register client.

OSStatus **CSRegisterTimer** (
PCCardClientID registeredClientID,
PCCardTimerID* lpNewTimerID,
long delay)

Parameters:

→ `registerClientID`	- "ClientHandle" from CSRegisterClient call.
← `lpNewTimerID`	- "TimerClientHandle"
→ `delay`	- number of milliseconds to expire before the registered callback is called.

Return Codes:

noErr	- if socket and function numbers are valid
paramErr	- if the parameters are not valid
kBadSocketErr	- if the socket number is invalid

8.9.2.7 CSDeregisterTimer

CSDeregisterTimeris provided to unregister timer.

OSStatus **CSDeregisterClient**(PCCardTimerID clientID)

Parameters:

→ `clientID`	- "TimerClientHandle"

Return Codes:

noErr	- if socket and function numbers are valid
kBadClientIDErr	- if the client ID is invalid

8.9.2.8 CSNotifyClients

CSNotifyClients will look through all of the registered clients and will execute the client callback routines for all clients which are registered for this socket and which have an event mask that includes the event.

If any client callback returns anything other than noErr, no further client callbacks will be executed and CSNotifyClients will return that error code,

OSStatus **CSNotifyClients** (

PCCardSocket	vSocket,
PCCardEvent *	theEvent)

Parameters:

→ `vSocket`	- the virtual socket
→ `theEvent`	- the Event bit(s) Table C.1

Return Codes:

noErr	- if all client callbacks returned noErr (or there were no clients registered)

8.9.2.9 CSGetStatus

The GetStatus control call returns the current status of a PC Card.

OSStatus **CSGetStatus** (

PCCardSocket	vSocket,
PCCardSocketStatus *	currentState,
PCCardSocketStatus *	changedState,
PCCardVoltage *	Vcc,
PCCardVoltage *	Vpp)

Parameters:

→ `socket`	- Virtual socket number
← `currentState`	- Current state of the socket Table C.2
← `changedState`	- delta bits of the socket since last cleared Table C.2

← Vcc	- Vcc setting of the socket
← Vpp	- Vpp setting of the socket

Return Codes:

noErr	- if socket and function numbers are valid
kBadSocketErr	- if the socket is invalid

8.9.3 Window Services Interface

8.9.3.1 CSRequestWindow

The RequestWindow call assigns a range of system address space to a PC Card of the specified Device.

When addressing the system memory range, the windowSpeed parameter is used as the requested access speed. The windowOffset parameter is the offset in the PC Card space. WindowOffset is adjusted by CSRequestWindow, if the requested offset is not on an alignment boundary. The windowBase is the host memory space and is returned to the calling program. The windowSize parameter is adjusted to the next largest alignment requirement, if needed. The memory window requested is immediately allocated and enabled, if no error is returned. The windowAttributes parameter must state that a memory window is being requested.

When an IO address range is requested, the IO range is only allocated and reserved, not enabled. The RequestConfiguration control call must be invoked to enable access to the IO range. The windowSpeed parameter is used to pass the number of I/O decode lines. The windowBase is the host I/O space and is returned to the calling program. The windowSize parameter is adjusted to the next largest alignment requirement, if needed. The windowAttributes parameter must state that an I/O window is being requested.

The requestedWindow parameter returns the window handle that is used in subsequent window calls. To release a memory window, call CSReleaseWindow. To release an I/O window the CSReleaseConfiguration call must be called. This ensures that the I/O windows are released and the socket interface is corrected, if needed.

OSStatus **CSRequestWindow**(

PCCardSocket	vSocket,
UInt32	device,
PCCardWindowAttributes	windowAttributes,
PCCardAccessSpeed	windowSpeed,
LogicalAddress *	windowBase,
PCCardWindowSize*	windowSize,
PCCardWindowOffset *	windowOffset,
PCCardWindowID *	requestedWindow)

Parameters:

→ `socket`	- Virtual socket number
→ `device`	- PC Card device number
↔ `windowAttributes`	- WindowState bitmask Table C.3
↔ `windowSpeed`	- Memory access speed or ioDecodeLines
← `windowBase`	- Window base address in bytes
↔ `windowSize`	- Minimum window size in bytes (input 16-bit only)
→windowOffset	- PC Card memory offset
← `requestedWindow`	- Window Identifier

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kBadSizeErr	- if the requested window size cannot be accommodated
kBadSpeedErr	- if the requested access speed is invalid or cannot be accommodated
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated

8.9.3.2 CSReleaseWindow

The ReleaseWindow call disables the memory window assigned to the adapter and deallocates the host memory space previously assigned to a PC Card by the RequestWindow control call. This call only releases memory windows, not I/O windows.

OSStatus **CSReleaseWindow**(
PCCardWindowID windowToRelease)

Parameters:

→ `windowToRelease`	- Window Handle

Return Codes:

noErr	- if window handle is valid
kBadHandleErr	- if window handle is invalid
kConfigurationLockedErr	- if the PC Card function is already configured (IO windows only)

8.9.3.3 CSModifyWindow [16-bit PC Card Only]

The ModifyWindow control call allows the Access Speed and/or Card Offset of a 16-bit PC Card memory window to be modified.

OSStatus **CSModifyWindow**(

PCCardWindowID	windowToModify,
PCCardWindowAttributes	windowAttributes,
PCCardAccessSpeed	memorySpeed,
PCCardWindowOffset*	windowOffset)

Parameters:

→ `windowToModify`	- Window Identifier
→ `windowAttributes`	- WindowState bitmask Table C.3
→ `memorySpeed`	- Requested memory access speed
→ `windowOffset`	- PC Card memory offset

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadHandleErr	- if window handle is invalid
kBadAttributeErr	- if any attributes are invalid
kBadSpeedErr	- if the requested access speed is invalid or cannot be accommodated
kBadOffsetErr	- if the card offset is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.4 Configuration Services

8.9.4.1 CSRequestConfiguration

The RequestConfiguration control call configures the requested device on the PC Card. This call also sets up the socket interface to be a either an I/O or memory type. This is controlled by the socketInterface parameter. The configRegPresentMask states which configRegValues are present. The configRegValues relates to the Function Configuration Register(s) on the PC Card, see electrical specification.

All IO windows previously assigned by RequestWindow are enabled for access. Any DMA channel previously assigned by RequestDMA is enabled if so specified in the attributes field.

OSStatus **CSRequestConfiguration** (

PCCardSocket	vSocket,
UInt32	device,
PCCardConfigOptions	configOptions,
PCCardInterfaceType	socketInterface,
PCCardInterfaceID	customInterface,
PCCardVoltage	vcc,
PCCardVoltage	vpp,
PCCardIRQ	IRQ,
PCCardDMA	DMA,
UInt32	configRegBaseAddress;
PCCardConfigPresentMask	configRegPresentMask,
PCCardFunctionConfigReg *	configRegValues)

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number
→ `configAttributes`	- Placed in function config option register Table C.4
→ `socketInterface`	-Interface type (I/O, Memory, custom) Table C.5
→ `customInterface`	- Custom interface type (zoom Video)
→ `vcc`	- Vcc voltage in tenths of volts
→ `vpp`	- Vpp voltage in tenths of volts
← `IRQ`	– kIRQEnable signifies that PC Card interrupts are enabled.
← `DMA`	– reserved for future use, not applicable to the Mac
→ `configRegBaseAddress`	- Base address for configuration
→ `configRegPresentMask`	- 16-bit Card config register bitmap
→ `configRegValues`	- 16-bit Card config register values

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated
kBadTypeErr	- if the interface type is invalid
kBadVccErr	- if Vcc is invalid or unsupported
kBadVppErr	- if Vpp is invalid or unsupported
kBadArgsErr	- if the Custom ID is invalid
kConfigurationLockedErr	- if the PC Card function is already configured
kNoCardErr	- if no PC Card is present in the socket

8.9.4.2 CSReleaseConfiguration

The ReleaseConfiguration control call deconfigures the device on the PC Card. If no I/O configurations exist on the PC Card, the socket interface is placed back into a memory interface.

All IO windows previously assigned by RequestWindow are disabled. Any DMA channel previously assigned by RequestDMA and enabled by RequestConfiguration or ModifyConfiguration is disabled.

OSStatus **CSReleaseConfiguration** (

PCCardSocket	vSocket,
UInt32	device)

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number

Return Codes:

noErr	- if the socket and function numbers are valid and there is a configuration to release
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid

8.9.4.3 CSModifyConfiguration

The ModifyConfiguration control call allows a PC Card configuration to be modified without having to issue RequestConfiguration and ReleaseConfiguration calls.

OSStatus **CSModifyConfiguration**(

PCCardSocket	vSocket,
UInt32	device,
PCCardConfigOptions	modifyAttributes,
PCCardIRQ	IRQ,
PCCardDMA	DMA,
PCCardVoltage	vpp)

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number
→ `modifyAttributes`	- Modify configuration attributes Table C.4
← `IRQ`	– kIRQEnable signifies that PC Card interrupts are enabled.
← `DMA`	– reserved for future use, not applicable to the Mac
→ `vpp`	- Vpp voltage in tenths of volts

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kBadAttributeErr	- if any attributes are invalid, conflicting, or cannot be accommodated
kBadVppErr	- if Vpp is invalid or unsupported
kNoCardErr	- if no PC Card is present in the socket

8.9.4.4 CSReadConfigRegister

The ReadConfigRegister control call allows the PC Card device configuration registers to be read. The whichRegister parameter selects the function configuration register to be read.

OSStatus **CSReadConfigurationRegister** (

```
                    PCCardSocket            vSocket,
                    UInt32                  device,
                    PCCardConfigPresentMask whichRegister,
                    UInt32                  configRegBaseAddress,
                    UInt8 *                 value)
```

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number
→ `whichRegister`	- FCR register index
→ configRegBaseAddress	- FCR base address
← `value`	- Value read from FCR

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kBadArgsErr	- if the register type or memory register offset is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.4.5 CSWriteConfigRegister

The WriteConfigRegister control call allows the PC Card device configuration registers to be written. The whichRegister parameter selects the function configuration register to be read.

```
OSStatus CSWriteConfigurationRegister (
                    PCCardSocket            vSocket,
                    UInt32                  device,
                    PCCardConfigPresentMask whichRegister,
                    UInt32                  configRegBaseAddress,
                    UInt8                   value)
```

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number
→ `whichRegister`	- FCR register index
→ configRegBaseAddress	- FCR base address
→ `value`	- Value written to FCR

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kBadArgsErr	- if the register type or memory register offset

	is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.4.6 CSResetFunction

The ResetFunction control call resets the specified function. Clients are notified if the function reset can be preformed. If any one client rejects the reset request the reset will not occur. The client requesting the reset function must save the device's configuration or request a new configuration. Device notification is performed when the reset is complete.

OSStatus **CSResetFunction** (
PCCardSocket vSocket,
UInt32 device)

Parameters

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kInUseErr	- if the PC Card function is configured
kNoCardErr	- if no PC Card is present in the socket

8.9.5 CIS Services Interface

8.9.5.1 CSValidateCIS

The CSValidateCIS control call validates the Card Information Structure for the specified Socket and Device. It returns the number of valid tuple chains in the device CIS. If a checksum tuple is present it will validate the checksum.

OSStatus **CSValidateCIS**(
PCCardSocket vSocket,
UInt32 device,
UInt32 * cisChainCount)

Parameters:

→ `vSocket`	- Virtual socket number
→ `device`	- PC Card device number
← `cisChainCount`	- Number of valid tuple chains located in CIS

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid

kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.5.2 CSGetDeviceCount

The CSGetDeviceCount control call examines the Card Information Structure for the number of devices present on the PC Card. It returns the number of devices in deviceCount.

OSStatus **CSGetDeviceCount**(

PCCardSocket	vSocket,
UInt32 *	deviceCount)

Parameters

→ `vSocket`	- Virtual socket number
← `deviceCount`	- Number of devices on PC Card

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.5.3 CSGetFirstTuple

The CSGetFirstTuple control call allows the tuples of the Card Information Structure to be read. To retrieve a specific tuple, set desiredTuple to the specific tuple code. To read the first tuple in the CIS, set the desiredTuple to 0FFh. If no tuple data is required, tupleBufferSize must be set to zero. If tupleBufferSize is greater then zero, the tuple data will be placed into the tupleDataBuffer. The length of the tuple data returned is passed back in the tupleBufferSize. This call always initializes the tupleIterator to the beginning of the CIS before searching for tuples. The tupleIterator must be preserved for subsequent tuple reads.

OSStatus **CSGetFirstTuple**(

Socket	vSocket,
UInt32	device,
PCCardTupleIteratorPtr	tupleIterator,
Byte	desiredTuple,
void *	tupleData,
ByteCount *	tupleBufferSize,
Byte *	foundTuple)

Parameters

→ `vSUInt32ocket`	- Virtual socket number
→ `device`	- PC Card device number
↔ `tupleIterator`	- Card Services Internal Use Only
→ `desiredTuple`	- First occurrence of specific tuple code OR FFh for the first tuple in CIS

← `tupleData`	- Pointer to tuple data buffer
↔ `tupleBufferSize`	- Length of Tuple data buffer (in bytes) and length returned
← `foundTuple`	- tuple code found in the CIS

Return Codes:

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kNoMoreItemsErr	- if specified tuple was not found
kNoCardErr	- if no PC Card is present in the socket

8.9.5.4 CSGetNextTuple

The CSGetNextTuple control call allows the tuples of the Card Information Structure to be read. To retrieve a specific tuple, set desiredTuple to the specific tuple code. To read the next tuple in the CIS, set the desiredTuple to 0FFh. If no tuple data is required, tupleBufferSize must be set to zero. If tupleBufferSize is greater then zero, the tuple data will be placed into the tupleDataBuffer. The length of the tuple data returned is passed back in the tupleBufferSize. The next tuple examined is based on the CIS pointer that is stored in the tupleIterator. The CIS pointer is incremented to the next tuple, after a successful read.

```
OSStatus CSGetNextTuple(
        PCCardTupleIteratorPtr      tupleIterator,
        Byte                        desiredTuple,
        void *                      tupleData,
        ByteCount *                 tupleBufferSize,
        Byte *                      foundTuple)
```

Parameters:

↔ `tupleIterator`	- Card Services Internal Use Only
→ `desiredTuple`	- Next occurrence of specific tuple code OR FFh for the first tuple in CIS
← `tupleData`	- Pointer to tuple data buffer
↔ `tupleBufferSize`	- Length of Tuple data buffer (in bytes) and length returned
← `foundTuple`	- tuple code found in the CIS

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadDeviceErr	- if the device number is invalid
kBadSocketErr	- if the socket is invalid
kNoMoreItemsErr	- if specified tuple was not found
kNoCardErr	- if no PC Card is present in the socket

8.9.6 Miscellaneous Services

8.9.6.1 CSGetDeviceCount

The CSGetDeviceCount call returns the number if devices found on a card.

```
OSStatus CSGetDeviceCount(
        PCCardSocket            vSocket,
        UInt32 *                deviceCount)
```

Parameters:

→ vSocket	- virtual socket number
← deviceCount	- number of devices on the card in the socket

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.2 CSGetSocketDeviceFromIterator

The CSGetSocketDeviceFromIterator call returns the number if devices found on a card.

```
OSStatus CSGetSocketDeviceFromIterator(
        PCCardSocket *          vSocket,
        UInt32 *                device,
        PCCardTupleIteratorPtr  tupleIterator)
```

Parameters:

← vSocket	- virtual socket number
← device	- device number
→tupleIterator	- tuple iterator

Return codes

noErr	- if all parameters are valid and request can be serviced

8.9.6.3 CSCardEject

The CSCardEject call ejects the card in the particular socket.

```
OSStatus CSCardEject(
        PCCardSocket *          vSocket)
```

Parameters:

← vSocket	- virtual socket number

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.4 CSGetCardType

The CSGetCardType call returns the card type.

OSStatus **CSGetCardType**(
PCCardSocket vSocket,
PCCardHardwareType * cardType)

Parameters:

→ vSocket	- virtual socket number
← cardType	- card type, cardBus or 16bit card Table C.9

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.5 CSGetInterruptSetMember

The CSGetInterruptSetMember calls socket services to get the Interrupt Set Member and pass it back to the calling function.

OSStatus **CSGetInterruptSetMember**(
PCCardSocket vSocket,
InterruptSetMember * ISTMember)

Parameters:

→ vSocket	- virtual socket number
← ISTMember	- ISTMember number of socket services

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.6 CSSetInterrupt

The CSSetInterrupt calls socket services to enable the functional interrupts. This is called when the target drivert calls the enablefunction routine that PcCard 3.0 overrides so that Pc Card 3.0 knows when to route functional interrupts.

OSStatus **CSSetInterrupt**(
PCCardSocket vSocket,
Boolean IRQEnable)

Parameters:

→ vSocket	- virtual socket number
→ IRQEnable	- Boolean to turn functional interrupts off and on.

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.7 CSSetRingIndicate

The CSSetRingIndicate calls socket services to enableringIndicate functioanlalty of a modem card.

OSStatus **CSSetRingIndicate**(
PCCardSocket vSocket,
Boolean setRingIndicate)

Parameters:

→ vSocket	- virtual socket number
→ setRingIndicate	- Boolean to turn ring indicate off and on.

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.8 CSPowerManagement

The CSPowerManagement call is used by the system whne the system is powered down during sleep

OSStatus **CSPowerManagement**(
PCCardSocket vSocket,
PCCardPowerOptions powerManagementOptions)

Parameters:

→ vSocket	- virtual socket number

→ powerManagementOptions - a flag that tells the system what power state to place the device. The supported power states are kPCCardPowerOff, kPCCardLowPower[1],kPCCardPowerOn.

Return codes

noErr	- if all parameters are valid and request can be serviced
kBadSocketErr	- if the socket is invalid
kNoCardErr	- if no PC Card is present in the socket

8.9.6.9 CSReportStatusChange

The **CSReportStatusChange** is called by socket services to notify the expert of a status change interrupt.

OSStatus **CSReportStatusChange**(

const RegEntryRef *	adapterRef,
PCCardSocket	whichSocket,
PCCardSCEvents	statusChange,
PCCardSocketStatus	socketStatus)

Parameters:

→ `adapterRef`	- RegEntryRef of the adapter reporting the status change.
→ whichSocket	- The socket number that caused the status change interrupt.
→ statusChange	- status change interrupt that occured.
→ socketStatus	- The socket status at the time of calling.

Return codes

noErr	- if all parameters are valid and request can be serviced
kInvalidNodeErr	- if the RegEntryRef is not valid
paramErr	- if a parameter is not valid
kPostCardEventErr	- if an error occured during posting the event to the PCCard Expert.

1. At the present time kPCCardLowPower is not supported

8.10 Socket Services Plug-in Interface

The Socket Service interface is a stateless device driver that is responsible for accessing the PC Card Hardware. It consists of a Socket Service Plug-in interface and a hardware abstraction layer that handles common hardware functions. This will allow Socket Service Plug-ins to share the common hardware abstraction DLL while allowing machine specific differences to be handled in the plug-in.

8.10.1 Apple Specific Plug-in Interface

The plug-in architecture defined by the Mac OS 8 architecture specification defines the following standard interface for all plug-ins.

8.10.1.1 _SSValidateHardware

The *_SSValidateHardware* call is a standard Apple family programming interface to allow family plug-ins a chance to ensure that the hardware is really compatible with the driver.

OSStatus **_SSValidateHardware**(const RegEntryRef *deviceID)

Parameters

→ `deviceID` - RegEntry ID for the socket hardware

Return codes

noErr	- if all parameters are valid
-1	- Valid socket Hardware is not present

8.10.1.2 _SSInitialize

The *_SSInitialize* call is used by the family programming interface to the socket service driver globals and install it's interrupts. If *_SSInitialize* is called with *replacingOldDriver* equal to true it is assumed that the driver is being replaced and has saved data to the name registry to restore the state of the driver so that the new driver can restore the data.

OSStatus **_SSInitialize**(
RegEntryIDPtr deviceID,
Boolean replacingOldDriver);

Parameters

→ `deviceID` - RegEntry ID for the socket hardware
→ `replacingOldDriver` - boolean signifying if this is a replacement operation.

Return codes

noErr - if all parameters are valid
kGeneralFailureErr-

8.10.1.3 _SSSuspend

The *_SSSuspend* call will place the socket adapter into low power mode.

OSStatus **_SSSuspend**(
const RegEntryRef *deviceID);

Parameters

→ `deviceID` - RegEntry ID for the socket hardware

Return codes

noErr - if all parameters are valid

8.10.1.4 _SSResume

The *_SSResume* call will power the socket adapter back up after a *_SSSuspend*.

OSStatus **_SSResume**(
const RegEntryRef *deviceID)

Parameters

→ `deviceID` - RegEntry ID for the socket hardware

Return codes

noErr - if all parameters are valid

8.10.1.5 _SSFinalize

The *_SSFinalize* call will place the socket service adapter into a shutdown state. *_SSFinalize* when called with *replacingOldDriver* equal to true will save all the necessary data to the name registry so that a replacement driver take it's place.

OSStatus **_SSFinalize**(
RegEntryIDPtr deviceID,
Boolean replacingOldDriver);

Parameters

→ `deviceID` - RegEntry ID for the socket hardware
→ `replacingOldDriver` - boolean signifying if this is a replacement operation.

Return codes

noErr - if all parameters are valid
kGeneralFailureErr-

8.10.2 Adapter Specific Interface

8.10.2.1 _SSInquireAdapter

The *SSInquireAdapter* control call returns the hardware capabilities of the adapter.

```
OSStatus _SSInquireAdapter (
        PCCardSocket *          socket,
        PCCardWindow*           window,
        PCCardWindow *          bridgeWindow,
        PCCardAdapterCapabilities *CMask);
```

Parameters

←`socket` - number of sockets
←`windows` - number of windows
←`bridgeWindow` - number of bridge windows
←`CMask` - bit-mask used to return the capabilities of the adapter
Table C.7

Return codes

noErr - if adapter is valid
kBadAdapterErr - if adapter is invalid

8.10.3 Socket Specific Interface

8.10.3.1 _SSInquireSocket

The *_SSInquireSocket* control call returns the capabilities of a socket.

The card Events bit-mask indicates the card events that can be generated by the socket.

OSStatus **_SSInquireSocket**(
PCCardSocket socket,
PCCardWindow * window,
PCCardSocketStatus *socketStatus,
PCCardSCEvents * cardEvents);

Parameters

→`socket`	- socket number (0 based)
←`window`	- number of windows
←`socketStatus`	- bit-mask of supported status bits Table C.2
←`cardEvents`	- bit-mask of supported card events (these can trigger status change events) Table C.8

Return codes

noErr	- if socket is valid
kBadSocketErr	- if socket is invalid

8.10.3.2 _SSGetSocket

The *_SSGetSocket control* call returns the current parameter settings of a socket.

OSStatus **_SSGetSocket**(
PCCardSocket socket,
PCCardVoltage * vcc,
PCCardVoltage * vpp,
PCCardVoltage * vs,
PCCardInterfaceType *socketIF,
PCCardCustomInterfaceID *customIFID,
PCCardSocketStatus *socketStatus,
PCCardSCEvents * cardEvents,
PCCardIRQ * IRQ,
PCCardDMA * DMA);

Parameters

→ `socket`	- socket number (0 based)
← `vcc`	- supply voltage
← `vpp`	- programming voltage
← `vs`	- voltage sense
← `socketIF`	- ioIF, memoryIF, customIF Table C.5
← `customIFID`	- ZoomVideoID, etc...
← `socketStatus`	- Status of the socket Table C.2
← `cardEvents`	– bit-mask of enabled card events

Table C.8

← `IRQ` – kIRQEnable signifies that PC Card interrupts are enabled.

← `DMA` – reserved for future use, not applicable to the Mac

Return codes

noErr - if socket is valid
kBadSocketErr - if socket is invalid

8.10.3.3 _SSSetSocket

The *_SSSetSocket* control call programs socket parameters. The driver avoids reprogramming a parameter when the new value matches the current setting.

The cardEvents parameter specifies which card events are enabled.

A vcc value of zero is interpreted as a request to remove (power down) the socket; the driver removes power from both the vcc and vpp pins.

Note: The Socket Services Plug-in will always enable the RingIndicate capabilities of the socket. It is the responsibility of the Power Management Utility to enable/disable it from it's hardware.

```
OSStatus _SSSetSocket(
        PCCardSocket            socket,
        PCCardVoltage           Vcc,
        PCCardVoltage           Vpp,
        PCCardInterfaceType     socketIF,
        PCCardCustomInterfaceID customIFID,
        PCCardSCEvents          cardEvents,
        PCCardIRQ *             IRQ,
        PCCardDMA *             DMA);
```

Parameters

→ `socket` - socket number (0 based)
→ `Vcc` - supply voltage
→ `Vpp` - programming voltage
→ `socketIF` - ioIF, memoryIF, customIF
Table C.5
→ `customIFID` - ZoomVideoID, etc...
→ `cardEvents` - bit-mask of enabled card events
Table C.8
→ `IRQ` – kIRQEnable signifies that PC Card interrupts are enabled.
→ `DMA` – reserved for future use, not applicable to the

Return codes

noErr	- if socket is valid
kBadSocketErr	- if socket is invalid
kCardBusCardErr	- if the Card is a CardBus card (only until CardBus is implemented)
kBadTypeErr	- if IFType not supported
kBadVccErr	- if Vcc level is invalid
kBadVppErr	- if Vpp1 or Vpp2 level is invalid

8.10.3.4 _SSResetSocket

The *_SSResetSocket* control call resets the PC CARD hardware.

OSStatus **_SSResetSocket** (PCCardSocket socket);

Parameters:

→ `socket`	- socket number (0 based)

Return codes

noErr	- if socket is valid
kBadSocketErr	- if socket is invalid
kCardBusCardErr	- if the Card is a CardBus card (only until CardBus is implemented)

8.10.3.5 _SSGetStatus

The *_SSGetCardStatus* control call returns a bit-mask of the current status associated with a PC CARD.

OSStatus **_SSGetStatus**(
PCCardSocket socket,
PCCardSocketStatus * socketStatus);

Parameters

→ `socket`	- socket number (0 based)
←`socketStatus`	- socket status bit-mask Table C.2

Return codes

noErr	- if socket is valid
kBadSocketErr	- if socket is invalid

8.10.4 Window Services Specific Interface

8.10.4.1 _SSInquireWindow

The *_SSInquireWindow* control call returns the hardware capabilities of a window. WindowType is an indication of the mapping capabilities of the window. DataWidth is a bit-mask representing the data sizing capabilities of the window. The window number is always an absolute window.

```
OSStatus _SSInquireWindow(
        PCCardSocket *        socket,
        PCCardWindow          window,
        PCCardWindowState *windowState,
        PCCardWindowSize *windowSize,
        PCCardWindowAlign *windowAlign);
```

Parameters

← `socket`	- socket number (0 based)
→ `window`	- window number (0 based)
← `windowState`	- Type:I/O, common/attribute - width:8,16,32 bits - enabled:window may be disabled and enabled without reprogramming it's characteristics write protect: window supports write protect Table C.3
← `windowSize`	- maximum window size (bytes)
← `windowAlign`	- window alignment boundary

Return codes

noErr	- if socket is valid
kBadWindowErr	- if window is invalid
kCardBusCardErr	- if a CardBus PC Card is present

8.10.4.2 _SSGetWindow

The *_SSGetWindow* control call returns the current parameter settings of a window. The window number is always an absolute window.

```
OSStatus _SSGetWindow(
        PCCardSocket *        socket,
        PCCardWindow          window,
        PCCardWindowState *windowState,
        LogicalAddress *      startAddress,
        PCCardWindowSize *windowSize,
        PCCardWindowOffset *windowOffset,
        PCCardAccessSpeed *memSpeed);
```

Parameters

← `socket` - socket number (0 based)
→ `window` - window number (0 based)
← `windowState` - Type:I/O, common/attribute
width:8,16,32 bits
enabled:enabled state of the window
write protect: write protect state of the window
Table C.3
← `startAddress` - host base logical address
← `windowSize` - window size (bytes)
← `windowOffset` - PC Card window offset address
← `memSpeed` - memory access speed

Return codes

noErr - if socket is valid
kBadWindowErr - if window is invalid
kCardBusCardErr - if a CardBus PC Card is present

8.10.4.3 _SSSetWindow

The *_SSSetWindow* control call programs a windows parameters.

If the adapter does not support the requested access speed, the driver selects the fastest access speed that does not exceed the requested speed.

```
OSStatus _SSSetWindow(
        PCCardSocket        socket,
        PCCardWindow        window,
        PCCardWindowState   windowState,
        LogicalAddress      startAddress,
        PCCardWindowSize    windowSize,
        PCCardWindowOffset  windowOffset,
        PCCardAccessSpeed   memSpeed);
```

Parameters

→ `socket` - socket number (0 based)
→ `window` - window number (0 based)
→ `windowState` - Type:I/O, common/attribute
width:8,16,32 bits
enabled:enabled state of the window
write protect: write protect state of the window
Table C.3
↔ `startAddress` - host base logical address
→ `windowSize` - window size (bytes)
→ `windowOffset` - PC Card window offset address
→ `memSpeed` - memory access speed

Return codes

noErr - if all parameters are valid
kBadSocketErr - if socket is invalid for window

kBadWindowErr - if window is invalid
kBadAttributeErr - if requested State does not match the windows capabilities
kBadBaseErr - if the start address is not valid
kBadSizeErr - if size is invalid
kBadSpeedErr - if memSpeed is invalid
kBadOffsetErr - If offset address is bad
kCardBusCardErr - if a CardBus PC Card is present

8.10.4.4 _SSGetWindowOffset

The *_SSGetWindowOffset* control call returns the current configuration of the memory window by the input parameters. The window number is always an absolute window.

OSStatus **_SSGetWindowOffset**(
PCCardSocket socket,
PCCardWindow window,
PCCardWindowState *windowState,
PCCardWindowOffset *windowOffset);

Parameters

← `socket`	- socket number (0 based)
→ `window`	- window number (0 based)
← `windowState`	- state of the window, write-protected, enabled etc... Table C.3
← `windowOffset`	- PC CARD window offset address

Return codes

noErr - if all parameters are valid
kBadSocketErr - if socket is invalid for window
kBadWindowErr - if window is invalid
kCardBusCardErr - if a CardBus PC Card is present

8.10.4.5 _SSSetWindowOffset

The *_SSSetWindowOffset* control call configures the page specified by the input parameters. It is only valid for memory windows. This will be unsupported for CardBus PC Cards. The window number is always an absolute window.

OSStatus **_SSSetWindowOffset**(
PCCardSocket socket,
PCCardWindow window,
PCCardWindowState windowState,
PCCardWindowOffset windowOffset);

Parameters

→ `socket` - socket number (0 based)
→ `window` - window number (0 based)
→ `windowState` - state of the window, write-protected, enabled etc... Table C.3
→ `windowOffset` - PC CARD window offset address

Return codes

noErr - if all parameters are valid
kBadSocketErr - if socket is invalid for window
kBadWindowErr - if window is invalid
kBadAttributeErr - if state is invalid
kBadOffsetErr - if offset is invalid
kCardBusCardErr - if a CardBus PC Card is present

8.10.5 CardBus Specific calls

The following calls are specific to CardBus controllers and are not required for PC Card 16 controllers.

8.10.5.1 _SSWriteConfigurationSpace

The _SSWriteConfigurationSpace call allows card services to write to a CardBus card's configuration space.

```
OSStatus _SSWriteConfigurationSpace(
        const RegEntryRef*  deviceID,
        PCCardSocket        socket,
        UInt32              device,
        UInt32              offset,
        void*               data,
        UInt32              size);
```

Parameters

→ `deviceID` - Device identifier
→ `socket` - socket number (0 based)
→ `device` - device number (0 based)
→ `offset` - offset into configuration space
→ `data` - data buffer pointer
→ `size` - size of data to be written

Return codes

noErr - if all parameters are valid
kBadOffsetErr - if offset is invalid
kBadSocketErr - if socket is invalid for window
k16BitCardErr - if the card present is a 16-bit card

8.10.5.2 _SSReadConfigurationSpace

The _SSReadConfigurationSpace call allows card services to read from the CardBus card's configuration space.

```
OSStatus _SSReadConfigurationSpace(
        const RegEntryRef*   deviceID,
        PCCardSocket         socket,
        UInt32               device,
        UInt32               offset,
        void*                data,
        UInt32               size);
```

Parameters

→ `deviceID`	- Device identifier
→ `socket`	- socket number (0 based)
→ `device`	- device number (0 based)
→ `offset`	- offset into configuration space
→ `data`	- data buffer pointer
→ `size`	- size of data to be read

Return codes

noErr	- if all parameters are valid
kBadOffsetErr	- if offset is invalid
kBadSocketErr	- if socket is invalid for window
k16BitCardErr	- if the card present is a 16-bit card

8.10.6 Bridge Services Specific Interface

8.10.6.1 _SSInquireBridgeWindow

The *_SSInquireBridgeWindow* service returns information about the capabilities of the bridge window specified by the input parameters. The window number is always an absolute window.

```
OSStatus _SSInquireBridgeWindow(
        PCCardSocket *       socket,
        PCCardWindow         window,
        PCCardWindowState *windowState,
        PCCardWindowSize *windowSize,
        PCCardWindowAlign *windowAlign);
```

Parameters

← `socket`	- socket number (0 based)
→ `window`	- window number (0 based)
← `windowState`	- type:I/O, memory Table C.3
← `windowSize`	- maximum window size in bytes
← `windowAlign`	- window alignment boundary

Return codes

noErr	- if all parameters are valid
kBadWindowErr	- if window is invalid
kBadAttributeErr	- if state is invalid
kBadOffsetErr	- if offset is invalid
kBadSocketErr	- if socket is invalid for window

8.10.6.2 _SSGetBridgeWindow

The *_SSGetBridgeWindow* service returns information about the current configuration of the bridge window. The window number is always an absolute window.

```
OSStatus _SSGetBridgeWindow(
        PCCardSocket *          socket,
        PCCardWindow            window,
        PCCardWindowState *windowState,
        LogicalAddress *        startAddress,
        PCCardWindowSize *windowSize);
```

Parameters

← `socket`	- socket number (0 based)
→ `window`	- window number (0 based)
← `windowState`	- state of the window, IO, enabled, prefetch cachable Table C.3
← `startAddress`	- base address of bridge window
← `windowSize`	- size of the bridge window in bytes

Return codes

noErr	- if all parameters are valid
kBadWindowErr	- if window is invalid
kBadAttributeErr	- if state is invalid
kBadOffsetErr	- if offset is invalid
k16BitCardErr	- if the card present is a 16-bit card

8.10.6.3 _SSSetBridgeWindow

The *_SSSetBridgeWindow* service returns information about the current configuration of the bridge window

```
OSStatus _SSSetBridgeWindow(
        PCCardSocket            socket,
```

PCCardWindow window,
PCCardWindowState windowState,
LogicalAddress startAddress,
PCCardWindowSize windowSize);

Parameters

→ `socket` - socket number (0 based)
→ `window` - window number (0 based)
→ `windowState` - state of the window, IO, enabled
Table C.3
→ `startAddress` - host start address of the bridge window
→ `windowSize` - size of bridge window in bytes

Return codes

noErr - if all parameters are valid
kBadWindowErr - if window is invalid
kBadAttributeErr - if state is invalid
k16BitCardErr - if the card is a 16-bit card
kBadBaseErr - if the base address is invalid
kBadSizeErr - if the size is invalid

8.10.7 Platform Specific Service Interface

The following calls are handled directly by the Socket Service Plug-in since there may be specific system hardware algorithms that will be applied to complete the calls.

8.10.7.1 _SSEjectCard

The *_SSEjectCard* control call physically ejects a card from a socket.

OSStatus **_SSEjectCard**(PCCardSocket socket);

Parameters

→ `socket` - socket number (0 based)

Return codes

noErr - if all parameters are valid
kBadSocketErr - if Socket is invalid
kNoCardErr - if socket is empty
kGeneralFailureErr- if card did not successfully eject

8.10.7.2 _SSGetInterruptSetMember

The *_SSGetInterruptSetMember* call is used to retrieve the interrupt SetID and member of the card interrupt set that the socket service driver created at initialization.

OSStatus **_SSGetInterruptSetMember** (
PCCardSocket socket,
InterruptSetMember * ISTMember)

Parameters

→ `socket`	- socket number (0 based)
← `ISTMember`	- Interrupt set member structure containing the SetID and member numbers

Return codes

noErr	- if all parameters are valid
kBadSocketErr	- if Socket is invalid

8.10.8 Interrupt Source Tree Construction

8.10.8.1 Socket Service Driver Initialization:

1. Get the InterruptSetMember (setID, member) for Set A, member from the Name Registry.

2. Get the current Interrupt Functions for this InterruptSetMember.

3. Invoke the current InterruptDisableRoutine (IDR).

4. Initialize the saved copy of interruptCount to nil. This interruptCount will later be used to determine when the ISR is re-invoked.

5. Install it's own InterruptServiceRoutine (ISR) in Interrupt Set A, member 1.

6. Create Interrupt Set B, which will contain 1 member for each socket the driver supports. This Interrupt Set needs to be created by the Socket Services Driver, because it knows how many sockets it supports, and therefore how many members to create. This Interrupt Set will be created with the kReturnToParentWhenNotComplete option. This will cause the Driver ISR to be re-invoked if it's child (a member of Interrupt Set B) returns kIsrIsNotComplete. This will allow the Driver to also return kIsrIsNotComplete if the Card ISR could not process the interrupt. If this option was not set, the ISR of the next member in Interrupt Set B (for the next Card) would automatically (and incorrectly) be invoked when the first member returned kIsrIsNotComplete.

7. Invoke the current InterruptEnableRoutine (IER).

8.10.8.2 Card Enabler Initialization:

1. Create the “card” node in the Name Registry.

2. Get the InterruptSetMember (setID, member) for the Card (socket). This is done via the Card Services function CSGetInterruptSetMember. Which calls the Socket Services function SSGetInterruptSetMember.

3. Get the current Interrupt Functions for this InterruptSetMember.

4. Invoke the current InterruptDisableRoutine (IDR).

5. Initialize the saved copy of interruptCount to nil. This interruptCount will later be used to determine when the ISR is re-invoked.

6. Install it’s own InterruptServiceRoutine (ISR) in the Interrupt Set.

7. Add a “driver-ist” property containing the card’s InterruptSetMember to the “card” node in the Name Registry.

8. Create Interrupt Set C, which will contain 1 member for each device (function) on the PC Card. This Interrupt Set needs to be created by the Card Enabler, because it knows how many devices the PC Card has, and therefore how many members to create. This Interrupt Set will be created with the kReturnToParentWhenNotComplete option. This will cause the Card ISR to be re-invoked if it’s child (a member of Interrupt Set C) returns kIsrIsNotComplete. This will allow the Card ISR to also return kIsrIsNot-Complete if there are no more Device ISR’s to invoke, or if it determined that the interrupt was for that specific Device ISR and the Device ISR was unable to successfully service the interrupt. If this option was not set, the next member in Interrupt Set C would automatically be invoked when the previous member returned kIsrIsNotComplete. This could cause the ISR for the next Device (function) to get invoked for an interrupt that was not generated by that device.

9. Create a driver-ist entry in the Name Registry for the Device node(s).

10. Invoke the current InterruptEnableRoutine (IER).

8.10.8.3 Interrupt Processing

1. An interrupt is generated, this invokes Interrupt Set A, member 1.

2. Interrupt Set A, member 1 is the Socket Service Driver ISR. It determine whether the interrupt is a Card Status Change interrupt or a Functional interrupt. If it is a Card Status Change Interrupt, the Socket Service Driver

ISR handles it and returns kIsrIsComplete. If it is a Functional Interrupt, the Driver ISR determines which socket caused the interrupt and returns that member number. This will invoke the returned member number of Interrupt Set B.

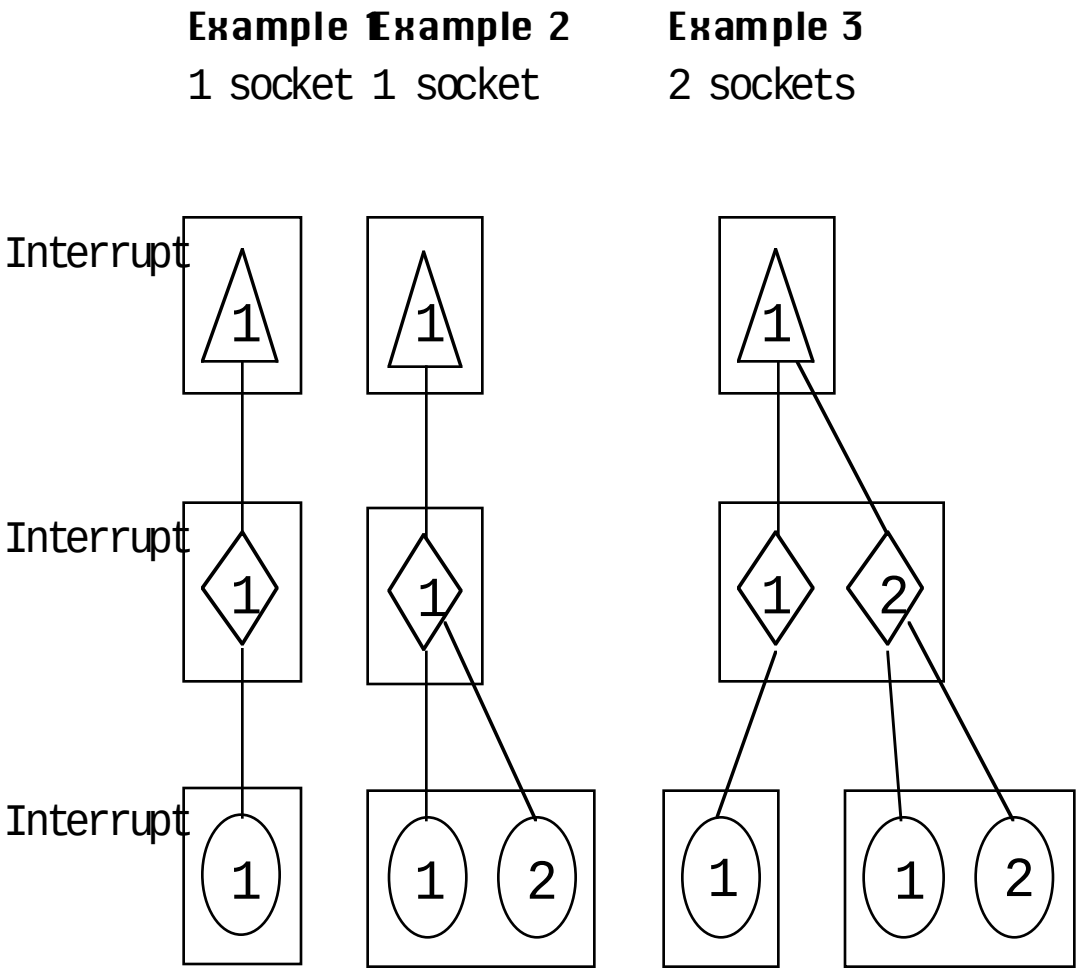
3. Interrupt Set B, member x is a Card (socket) ISR. It determines which device caused the interrupt and returns that member number. This will invoke the returned member number of Interrupt Set C.

4. Interrupt Set C, member x is a Device (function) ISR. It will attempt to service the interrupt and return kIsrIsComplete if successful, this will end interrupt processing. If the interrupt was not successfully serviced, the Device ISR will return kIsrIsNotComplete.

5. The parent member in Interrupt Set B is re-invoked if the Device ISR returned kIsrIsNotComplete. The Card ISR will then attempt to invoke the next child member (in Set C) by returning it's member number, or return kIsrIsNotComplete if there are no more members to invoke.

6. The parent member in Set A (the Driver ISR) is re-invoked if the Card ISR returned kIsrIsNotComplete. The Driver ISR will return kIsrIsNotComplete if it is invoked a second time while processing the same interrupt.

FIGURE 3. PC Card 3.0 IST Layout



9.0 Name Registry Properties for PC Cards

This section is broken up into three sections, controller, card and card functions. The node for the controllers will be built by either Open Firmware, the motherboard expert, or bus families. The card nodes are built by the PC Card expert. The device nodes will be built by the card enablers. All properties created by the PC Card family that are defined in the PC Card binding for Open Firmware should be created following that specification.

On machines that have Open Firmware, the PC Card family should check if the nodes have already been created. Since Open Firmware will have allocated physical hardware ranges, the family can either use them or free them and create its own. Since Open Firmware will not create all of the properties that we want the PC Card Family will have to at least add to a node. The PC Card family should not trust Open Firmware to set things up correctly, and it may have to fix things.

The purpose of Open Firmware and the card enablers is the same. The difference is that Open Firmware is for boot devices and enablers are for devices either don't need Open Firmware (not needed for boot) or have been hot plugged.

9.1 Socket Controller Node Properties

Property: **name**

Example: "TREX@12345678" or "pcixxxx,yyyy"

Source: Open Firmware, the motherboard expert or another bus family.?

Purpose: Standard prop-name to specify the implemented interface.

Property: **reg**

Example: see PC card OF binding

Source: Open Firmware, the motherboard expert or another bus family.?

Purpose: Standard prop-name to define the package's unit address.

Property: **assigned-addresses**

Example: -----

Source: Motherboard expert or another bus family.

Purpose: Assigned physical address ranges for the device

Property: **AAPL,addresses**

Example:

Source: Motherboard expert or another bus family.

Purpose: Provide LOGICAL addresses which corresponds to assigned-addresses property.

Property: **driver-ist**

Example: -----

Source: Motherboard expert or another bus family.

Purpose: Interrupt source tree node.

Property: **SocketNumber**

Example: 1

Source: PCCardSupportLibrary.

Purpose: Provide a virtual socket number.

9.2 Card Enabler Node Properties

Property: **name**

Example: “pccard104c,ac12”

Source: CISTPL_MANFID and/or CISTPL_VERS_1

Purpose: Standard prop-name to specify the implemented interface.

pccardVVVV,DDDD where VVVV is the manufactures id field and DDDD is the manufactures information field as defined below:

- VVVV string is defined by the Field ‘TPLMID_MANF’ in Tuple ‘CISTPL_MANFID’.

- DDDD string as defined by the Field ‘TPLMID_CARD’ in Tuple ‘CISTPL_MANFID’.

The VVVV and DDDD strings are ASCII hexadecimal, lower case, and without leading zeros.

If no CISTPL_MANFID tuple is found, the string “pccard” should be used.

Property: **compatible**

Example: “AAPL,GenericPCCardPlugin”

Source: created by Expert

Purpose: Defines alternate name property values, can be used by DFM for matching card enablers. See the “matching” property.

The name “AAPL,GenericPCCardPlugin” is reserved for use by Apple Computer, Inc.

Property: **SocketNumber**

Example: 1

Source: created by Expert

Purpose: Defines which socket into which a card has been inserted.

Property: **PCCardNodeType**

Example: ‘pccc’

Source: created by Expert

Purpose: Defines that the node is a card handled by the PCCard Expert.

Property: **driver-ist**

Example: -----

Source: Created by Socket Services Library. Populated by Expert.

Purpose: Defines the interrupt enabler, disabler and handler functions. Note that the card level and device lever IST nodes are closely dependent. If a developer wishes to install an interrupt handler in the card IST, he should save the existing interrupt handler, call the handler at interrupt time (to ensure that the device node(s) are called, return the value from the handler and reinstall the handler when he removes his custom handler. Failure to do this may lead to unserviced interrupts.

9.3 Functional Node Properties

Property: **name**

Example: pc-uart

Source: Card Enabler, CIS

Purpose: Used to match target driver to function node of PC Card. The name should exactly match the name in the descriptor of the driver which supports the device.

Property: **compatible**

Example: -----

Source: Card Enabler

Purpose: Defines alternate driver names which support device.

Property: **PCCardNodeType**

Example: ‘pccd’

Source: Card Enabler

Purpose: Demonstrates that the node is handled by a card enabler.

Property: **assigned-addresses**

Example: -----

Source: Card Enabler

Purpose: Encoded physical address ranges for the device. This property defines the type of memory access, size and location of the addresses defined in the AAPL,address property and is comprised of an array of PCIAssignedAddress structures.

Property: **AAPL,address**

Example: ----

Source: Created by Card Enabler

Purpose: Provide LOGICAL addresses which corresponds to assigned-addresses property. These are the addresses one reads and writes to access memory locations on the PC Card. The type of memory defined by these

addresses are described in the assigned-addresses property above. This property is comprised of an array of 32 bit addresses.

Property: **driver-ist**

Example: -----

Source: Created by Card Enabler

Purpose: Interrupt source tree node.

Note that the interrupt enabler and disabler functions present in the driver-ist are closely related and dependent on the card node driver-ist and should be saved and replaced by custom interrupt enabler/disabler functions. The existing interrupt enabler/disabler functions should be called by any custom enabler/disabler functions to preserve default behavior.

Property: **DeviceNumber**

Example: 8

Source: Created by Card Enabler

Purpose: 0 based index of device on card.

Property: **SocketNumber**

Example: 1

Source: Created by Card Enabler

Purpose: 0 based index of virtual socket number of card.

Property: **card services windows**

Example: -----

Source: Created by Card Enabler

Purpose: List of window handles allocated on card configuration. Used to release windows on card removal.

Property: **device-configured**

Example: -----

Source: Created by Card Enabler

Purpose: Defines that a device needs to release configuration on card removal.

Property: **16bitcard**

Example: -----

Source:

Purpose: prop-name, exists if the node implements the 16-bit PC Card interface.

Property: **CardBus**

Example: -----

Source: CISTPL_FUNCID, etc.

Purpose: prop-name, exists if the node implements CardBus.

Property: **device_type**

Example: “pccard-serial”

Source: Card enabler

Purpose: devices the type of device described.

Property: **device-id**

Example: “pccard-serial”

Source: Card enabler

Purpose: Defines the vendor ID of the device.

Property: **revision-id**

Example: 00000005

Source: Card enabler

Purpose: Defines the revision of the card as per the Vers_1 tuple.

Appendix C Data and Bit-Mask Definitions

The following appendix contains the bit-mask defined by the PC Card 3.0 implimentation

C.1 PC Card Events (PCCardEvents and PCCardEventMask)

TABLE 1. Registered Client PCCard Events (interestingEvents)

Event Name	Description
kPCCardInsertionMessage	card has been inserted into the socket
kPCCardRemovalMessage	card has been removed from the socket
kPCCardEjectionRequestMessage	user or other client is requesting a card ejection
kPCCardEjectionFailedMessage	eject failure due to electrical/mechanical problems
kPCCardPMResumeMessage	power management resume
kPCCardPMSuspendMessage	power management suspend
kPCCardPMSuspendRequest	power management sleep request
kPCCardPMSuspendRevoke	power management sleep revoke
kPCCardResetRequestMessage	physical reset has been requested by a client
kPCCardResetCompleteMessage	reset has completed
kPCCardBatteryDeadMessage	battery is no longer usable, data will be lost
kPCCardBatteryLowMessage	battery is weak and should be replaced
kPCCardWriteProtectMessage	card is now write protected
kPCCardWriteEnabledMessage	card is now write enabled
kPCCardTimerExpiredMessage	message sent when requested time has expired
kPCCardNullMessage	no messages pending (not sent to clients)
kPCCardLockMessage	card is locked into the socket with a mechanical latch
kPCCardUnlockMessage	card is no longer locked into the socket
kPCCardReadyMessage	card is ready to be accessed (not sent to clients)
kPCCardResetMessage	physical reset has completed (not sent to clients)

TABLE 1. **Registered Client PCCard Events (interestingEvents)**

Event Name	Description
kPCCardInsertionRequestMessage	request to insert a card using insertion motor (not sent to clients)
kPCCardInsertionCompleteMessage	insertion motor has finished inserting a card (not sent to clients)
kPCCardEjectionCompleteMessage	card ejection succeeded- do not touch hardware!
kPCCardResetPhysicalMessage	physical reset is about to occur on this card (not sent to clients)
kPCCardClientInfoMessage	client is to return client information (not sent to clients)
kPCCardSSUpdatedMessage	AddSocketServices/ReplaceSocket services has changed SS support (not sent to clients)
kPCCardFunctionInterruptMessage	card function interrupt (not sent to clients)
kPCCardAccessErrorMessage	client bus errored on access to socket (not sent to clients)
kPCCardUnconfiguredMessage	a CARD_READY was delivered to all clients and no client (not sent to clients)
kPCCardStatusChangedMessage	requested a configuration for the socket (not sent to clients)
kPCCardRequestAttentionMessage	(not sent to clients)
kPCCardEraseCompleteMessage	(not sent to clients)
kPCCardRegistrationCompleteMessage	(not sent to clients)

C.2 Socket Status Bit definitions (PCCardSocketStatus)

TABLE 2. **Socket status bit definitions**

Socket State	Description
kSTBatteryDead	battery dead
kSTBatteryLow	battery low
kSTBatteryGood	battery good
kSTPower	power is applied
kST16bit	16-bit PC Card present

TABLE 2. **Socket status bit definitions**

Socket State	Description
kSTCardBus	CardBus PC Card present
kSTMemoryCard	memory card present
kSTIOCard	I/O card present
kSTNotACard	unrecognizable PC Card detected
kSTWriteProtect	card is write-protected
kSTRingIndicate	ring indicator is active
kSTReady	ready
kSTDataLost	data may have been lost due to card removal
kSTReserved	Reserved

C.3 Window Attributes (PCCardWindowAttributes)

TABLE 3. **Window attribute bit-mask definitions**

Window Attribute	Description
kWSCommon	common memory window
kWSAttribute	attribute memory window
kWSIO	I/O window
kWSTypeMask	window type mask
kWS8bit	8-bit data width window
kWS16bit	16-bit data width window
kWS32bit	32-bit data width window (cardbus only)
kWSWidthMask	window data width mask
kWSEnabled	window enabled
kWSProtected	window write protected
kWSPrefetchable	bridge window prefetchable (CardBus only)
kWSCardBus	CardBus bridge window
kWSAutoSize	auto-size data width window

TABLE 3. Window attribute bit-mask definitions

Window Attribute	Description
kWSPageShared	page register is shared
kWSWindowSizeOffset	Used by CSModifyWindow only
kWSChangeAccessSpeed	Used by CSModifyWindow only

C.4 Configuration Attributes (PCCardConfigOptions)

TABLE 4. Configuration Attributes

Configuration attributes	description
kReservedBit0	Reserved
kEnableIRQSteering	Enable IRQ steering (not used on the Mac OS implementation)
kIRQChangeValid	IRQ Change valid (not used on the Mac OS implementation)
kReservedBit3	Reserved
kVppChangeValid	Vpp1 change valid
kReservedBit5	Reserved
kEnableDMAChannel	Enable DMA channel (not used on the Mac OS implementation)
kDMAChangeValid	DMA change valid (not used on the Mac OS implementation)
kReservedBit8	Reserved
kVSOverride	Vs override valid

C.5 Interface Types (PCCardInterfaceType)

TABLE 5. Interface types

Interface types	description
kIFTypeMask	IO & memory type mask
kIFCardBus	if bits 0 & 1 are zero then cardbus interface
kIFMemory	if bit 0 set memory IF
kIFIO	if bit 1 set IO IF
kIFReserved	bits 0 and 1 set is reserved
kIFDMA	if bit 3 set DMA supported
kIFVSKey	if bit 4 set supports low Voltage key
kIF33VCC	if bit 5 set socket supports 3.3v
kIFXXVCC	if bit 6 set socket supports X.X voltage

C.6 Supported device types and SubTypes (PCCardDevType and PCCardSubType)

TABLE 6. Supported device types (PcCardDevType/PCCardSubType)

Device type	description
kPCCardUnknownDeviceType	unknown device
kPCCardMultiFunctionType	Multi-Function device
kPCCardVideoAdaptorType	display device
kPCCardFixedDiskType	Fixed disk
kPCCardUnknownFixedDiskType	unknown Fixed disk type
kPCCardNetworkAdaptorType	Network device
kPCCardSerialPortType	Serial device (modem)
kPCCardParallelPortType	Parallel device type

TABLE 6. **Supported device types (PcCardDevType/PCCardSubType)**

Device type	**description**
kPCCardMemoryType	Memory device (sram)
kPCCardAIMSType	AIMS device
kPCCardSCSIType	SCSI Device
kPCCardSerialOnlySubType	serial only device
kPCCardDataModemSubType	Data modem sub type
kPCCardFaxModemSubType	Fax modem sub type
kPCCardFaxAndDataModemMask	Data Fax modem
kPCCardVoiceEncodingSubType	voice encoding type
kPCCardATAInterfaceDiskSubType	ATA disk sub type
kPCCardRotatingDeviceSubType	Rotating mediam sub type
kPCCardSiliconDevice	Silicon device

C.7 Adapter capabilities mask (PCCardAdapterCapabilities

TABLE 7. **Adapter capability bit-mask values**

Capabilities	**Description**
SS_ADPT_FLG_IND	indicators for write-protect, card lock,battery status, busy status, and XIP are shared for all sockets
SS_ADPT_FLG_PWR	if set indicates that the sockets share the same power control
SS_ADPT_FLG_DBW	all windows on the adapter must use the same Data Bus Width
SS_ADPT_FLG_CARDBUS	all sockets are CardBus PC Card capable
SS_ADPT_FLG_DMA	the adapter has DMA capability bits for adapter power characteristics
SS_ADPT_FLG_V33	adapter supports 3.3 volt power to socket
SS_ADPT_FLG_V50	adapter supports 5.0 volt power to socket
SS_ADPT_FLG_V12	adapter supports 12 volt power to socket

C.8 Socket Event mask (PCCardSCEvents)

TABLE 8. Socket Event Bit-mask

Event	Description
kSCBatteryDead	battery dead
kSCBatteryLow	battery low
kSCReady	ready
kSCCardDetect	Card Detect Status Change
kSCCardEjected	Card Ejected
kSCStatusChange	PC Card Status Change Signal Asserted
kSCRingIndicate	PC Card Ring Indicate Signal Asserted

C.9 PC Card 3.0 Hardware types (PCCardHardwareType)

TABLE 9. Pc Card 3.0 Hardware types

Hardware types	Description
kPCCard16HardwareType	PC Card-16 hardware
kCardBusHardwareType	CardBus Hardware

Appendix D Card Service Mapping

D.1 Mapping to ‘classic’ Card and Socket Services

The PC Card Family supports a simplified version of the Card & Socket Services version of the API described in 5.1 Card Services PC Card Standard. The following sections group calls defined in The Standard with descriptions of how they are implemented in the PC Card Family.

Cosmetic differences are not mentioned Ñ such as the format of various attributes bitmap definitions. In other cases, bit-mapped parameters have been combined where practical and reserved fields have been eliminated since there are no backward compatibility issues.

One functional difference which applies to many of the calls is that Client handles are not used for any call. This is because management of clients is implemented by the family itself.

Also, all block memory services will be broken out into a separate library which is not available at this time.

D.2 Mappings to the PC Card Standard

The functions are listed here with descriptions of how they differ from the Intel Card Services PC Card Standard. However, cosmetic differences are not mentioned. Such differences would include the format of various attributes bitmap definitions. In other cases, bitmapped parameters have been combined where practical and reserved fields have been eliminated since there are no backward compatibility issues. Only functional differences are called out in this section.

One functional difference which applies to many of the calls, is that Client handles are not used for any call. In fact, Client registration is not required.

D.3 Functionally Equivalent

The following calls are implemented in a manner functionally equivalent to the PC Card Standard.

- GetStatus
- ReleaseConfiguration
- ReleaseWindow
- RequestDMA
- ValidateCIS

D.4 Tuple Functions

The following APIs have been combined into a single call GetTuple:

- GetTupleData

D.5 Block Memory Device Family

The following calls have been omitted from Card Service under the MacOS since they would normally be supported under the block memory device plug-in:

- CheckEraseQueue
- CloseMemory
- CopyMemory
- DeregisterEraseQueue
- GetFirstPartition
- GetFirstRegion
- GetNextPartition
- GetNextRegion
- OpenMemory
- ReadMemory
- RegisterEraseQueue
- RegisterMTD
- SetRegion
- WriteMemory

D.6 Client Registration

Client registration is used solely for registering event callback addresses and for specifying event masks to perform client event notification. The following calls are not used within the scope of registering clients in this model since Card Services clients are limited to event notification:

- GetClientInfo
- GetFirstClient
- GetNextClient
- ReleaseExclusive
- ReleaseSocketMask
- RequestExclusive
- RequestSocketMask

D.7 MacOS Environment

The following calls have been omitted because they have been identified as unnecessary on the MacOS. Many of these calls are IBM PC centric used to

work around deficiancies in operating system and are used as a debugging interface during development.

- AdjustResourceInfo
- GetCardServicesInfo
- GetConfigurationInfo
- GetFirstWindow
- GetNextWindow
- MapLogSocket
- MapLogWindow
- MapPhySocket
- MapPhyWindow
- RegisterTimer
- ReleaseDMA
- ReleaseIO
- ReleaseIRQ
- ReplaceSocketServices
- RequestIO
- RequestIRQ
- ReturnSSEntry

D.8 Not Relevant to Hardware

Several functions deal with hardware features that are not relevant to our hardware environment, or to features which have been obsoleted by CardBus.

- GetMemPage
- MapMemPage
- ModifyWindow
- RequestWindow

D.9 API Simplification

In several cases we have been able to replace functions with a simplified API.

- AccessConfigurationRegister
- AddSocketServices
- ModifyConfiguration
- RequestConfiguration
- ResetFunction