PC CARD STANDARD

Volume 5

Card Services Specification

REVISION HISTORY

Date	Card Services Specification Version	PC Card Standard Release	Revisions
11/92	2.0	PCMCIA 2.01	Initial release
07/93	2.1	PCMCIA 2.1/JEIDA 4.2	Added Services Editorial corrections
02/95	5.0	February 1995 (5.0) Release	Added support for CardBus PC Cards Added Windows 16-bit Bindings Added Services Editorial corrections
03/95	N/A	March 1995 (5.01) Update	None
05/95	N/A	May 1995 (5.02) Update	None
11/95	5.01	November 1995 (5.1) Update	Added support for Custom Interfaces Editorial corrections
05/96	5.02	May 1996 (5.2) Update	Editorial corrections
03/97	6.0	6.0 Release	Added support for Hot Dock/Undock Added Streamlined PC Card Configuration
04/98	6.1	6.1 Update	Added Win32 Bindings Editorial corrections
02/99	7.0	7.0 Release	Added Windows NT Bindings Added support for PC Card Memory Paging Editorial corrections
03/00	7.1	7.1 Update	Added Latency Timer access to AccessConfigurationRegister Noted July 1, 2000 removal of references to DMA (Direct Memory Access)
11/00	7.2	7.2 Update	Removed all references to Direct Memory Access (DMA)
04/01	8.0	8.0 Release	None

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

1.1 Purpose

This document describes the interface provided by Card Services which allows PC Cards and sockets to be shared by multiple clients. (See ***3.1.1*** *Hardware Layer (PC Cards, Sockets* and Adapters).) Clients are the programs that access Card Services and may be device drivers, configuration utilities or application programs. This specification is intended to be independent of the hardware that actually manipulates PC Cards and sockets.

1.2 Scope

This document is intended to provide enough information for software developers to

a) create a Card Services implementation on a host computer, and

b) create programs that access and use PC Cards and sockets in a host computer.

1.3 Related Documents

This section identifies documents related to the Card Services Interface Specification. Information available in the following documents is not duplicated within this document.

PC Card Standard Release 8.0 (April 2001), PCMCIA / JEITA
Volume 1. ***Overview and Glossary***
Volume 2. ***Electrical Specification***
Volume 3. ***Physical Specification***
Volume 4. ***Metaformat Specification***
Volume 5. ***Card Services Specification***
Volume 6. ***Socket Services Specification***
Volume 7. ***PC Card ATA Specification***
Volume 8. ***PC Card Host Systems Specification***
Volume 9. ***Guidelines***
Volume 10. ***Media Storage Formats Specification***
Volume 11. ***XIP Specification***

IBM-AT Technical Reference Manual, First Edition, March 1984, International Business Machines.

This Page Intentionally Left Blank

2. OVERVIEW

Card Services has two goals. First, to support the ability of PC Card-aware device drivers, configuration utilities, and application programs (known as clients) to share PC Cards, sockets, and system resources. Second, to provide a centralized resource for the common functionality required by these clients.

The Card Services interface is structured in a client/server model. Application programs, device drivers, and utility programs are the clients requesting services. Card Services is the server providing the services requested by clients. The Card Service interface specified in this document defines how the clients and server communicate.

Note: Throughout the remainder of this document clients may be referred to as client device drivers without specifically mentioning application and utility programs. This does not mean that only device drivers can use Card Services. Card Services may be used by any resident or transient program that observes the interface protocol defined in this specification.

This document is divided into several sections:

- Chapter 3 is a functional description of the Card Services interface. The overall architecture is discussed with special focus on the programming interface, service groupings, callback interfaces, reported events, Memory Technology Drivers, and how Card Services processes the Card Information Structure.
- Chapter 4 identifies several assumptions and constraints which apply globally to the Card Services interface. The reader should keep these points in mind while reviewing other sections of the specification.
- Chapter 5 discusses each of the services provided by Card Services in detail. The purpose of each service is described with its input and output parameters.
- The appendices contain the following:
    - Service Codes
    - Event Codes
    - Return Codes
    - Bindings
    - MTD Helper Services
    - Media Access Services
    - Argument Usage Reference
    - Client Callback Argument Usage
    - OS Critical Section Handling

This Page Intentionally Left Blank

3. FUNCTIONAL DESCRIPTION

3.1 Architecture

Safely using PC Cards and sockets in a non-conflicting manner involves the interaction of several hardware and software architectural layers.

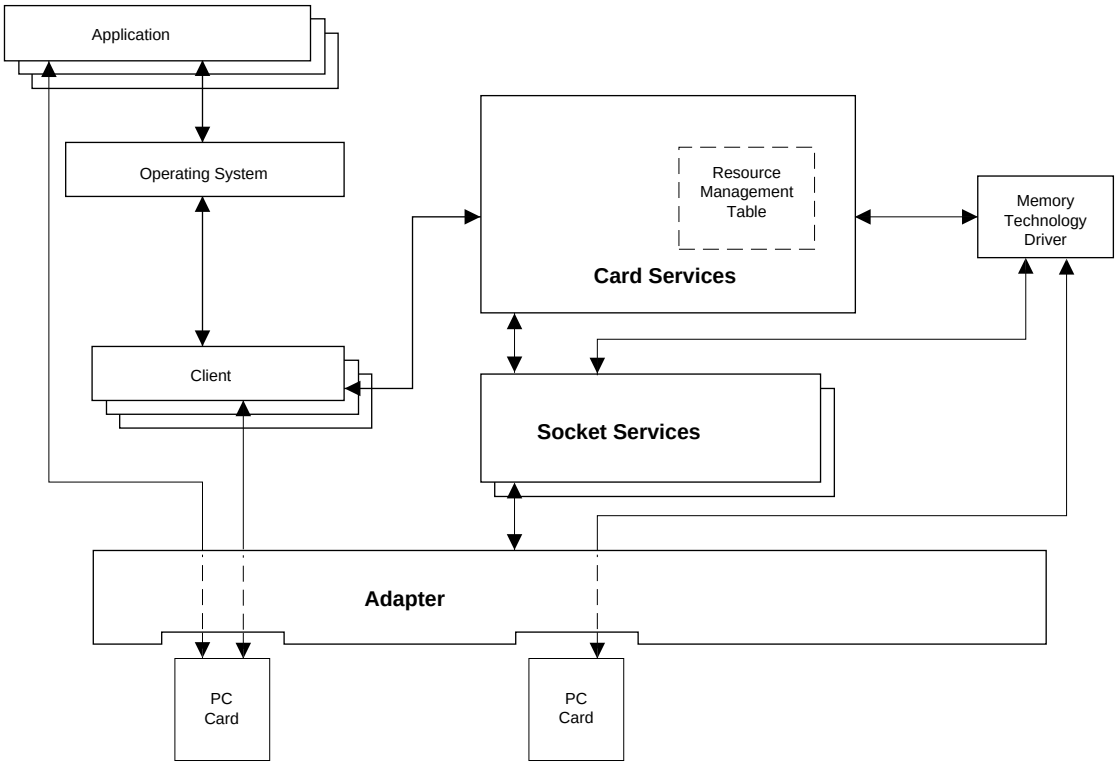

Figure 3-1: Software Architecture Diagram

3.1.1 Hardware Layer (PC Cards, Sockets and Adapters)

Cards compliant with the PC Card Standard are referred to as PC Cards. Originally the standard specified data storage or memory cards. Later releases of the standard expanded the definition of PC Cards to include peripheral expansion or I/O cards, provided for additional tuples, and further refined the software interface. This release specifies 3.3 volt operation, standard multiple function PC Cards and a 32-bit interface for all types of PC Cards.

All PC Cards have the same physical characteristics and compatible electrical characteristics.

PC Cards are plugged into sockets on a host system. Host systems may have one or more sockets and these sockets may be grouped together on one or more adapters. An example of a host system with more than one adapter would be one where an adapter was built into the motherboard and another plugged into the system's expansion bus.

Adapters usually generate maskable hardware interrupts when status changes occur in sockets or on PC Cards. Status changes include:

- card inserted or removed,
- battery low or dead,
- ejection or insertion request,
- card locked or unlocked in socket, and
- a busy to ready transition.

3.1.2 Socket Services

Immediately above the hardware layer the Socket Services software provides a standardized interface to manipulate PC Cards, sockets, and adapters. (See the ***Socket Services Specification***.)

As noted above, host systems may have more than one PC Card adapter present. Each adapter may have its own Socket Services handler. All instances of Socket Services are intended to support a single instance of Card Services. Card Services registers to receive notification of status changes on PC Cards or in sockets.

By making all accesses to adapters, sockets, and PC Cards through the Socket Services interface, higher-level software (including Card Services) is unaffected by different implementations of the hardware. Only a hardware-specific Socket Services implementation must be modified to accommodate any different hardware implementations.

3.1.3 Card Services

Above the Socket Services software layer is the Card Services layer. Card Services coordinates access to PC Cards, sockets and system resources among multiple clients. These clients may be resident or transient device drivers, system utilities, or application programs. There is only one Card Services implementation in a host system. (Unlike Socket Services where there may be multiple implementations to accommodate multiple adapters).

Card Services makes all access to the hardware layer through the Socket Services software interface. The single Card Services implementation is intended to be the sole client of all Socket Services implementations present. All Socket Services status change reporting is routed to this single Card Services implementation. Card Services then notifies interested clients when status changes occur.

To prevent conflicts with clients who are unaware of Card Services, direct access to the Socket Services interface is blocked by Card Services. A method of bypassing the Card Services blockage is provided for software developers of specialized applications which must access Socket Services. Programs which bypass Card Services and make direct access to Socket Services must ensure such access is benign and does not interfere with Card Services usage of Socket Services, PC Cards, sockets, or adapters.

Card Services preserves for its clients an abstract, socket-hardware-implementation independent view of a card and its resources. Card Services presents the same tuple organizational and resource allocation view to all of its clients whether the card is a 16-bit PC Card or a CardBus PC Card.

3.1.4 Memory Technology Drivers

The PC Card Standard supports a wide range of memory devices on PC Cards. While all PC Cards containing any such memory device may be read as if they contained static-RAM devices, special

programming algorithms may be required to write or erase the memory devices. Card Services hides the details of what is required to write or erase memory devices from client device drivers through byte-oriented write and copy services and a block-oriented erase service.

Within Card Services, Memory Technology Drivers (MTD) implement the specific programming algorithms required to access memory devices. These drivers may be embedded within Card Services or may register with a Card Services implementation at run-time. When PC Cards are installed, MTDs monitoring insertion events register with Card Services to support access to a memory device region through the Card Services read, write, copy, and erase services.

Card Services provides default MTDs for recognized regions. If Card Services recognizes a region as being composed of Static RAM devices, it installs a default MTD that supports read and write requests. Reads and writes are performed as simple memory accesses without any algorithmic operation. If Card Services recognizes a memory region but not the type of devices in the region, it installs a default MTD that supports read and write requests and fails erase requests. The reads and writes are performed as simple memory accesses without any algorithmic operation. Card Services may include MTD support for other device types that require specific programming algorithms. (See ***3.6 Memory Technology Drivers*** and see also ***Appendix-E, 10. MTD Helper Service Reference***.)

3.1.5 Client Device Drivers

Client device drivers refers to all users of Card Services. These may be device drivers, utility programs, or application programs.

3.2 Programming Interface

3.2.1 Calling Conventions

The Card Services interface uses a common set of conventions for all services.

3.2.1.1 Basic Operation

Card Services is invoked in a processor and Operating System dependent manner called a binding. (See ***Appendix-D, 9. Bindings.***)

All arguments for Card Services requests are passed in binding specific fashions. Card Services defines six generic arguments:

Service	**Status**
Handle	**Argument Length**
Pointer	**Argument Pointer**

Many Card Services requests pass all data in the **Service**, *Handle* and *Pointer* arguments. For such services, no argument packet (as referenced by *ArgPointer below*) is required. If a request requires more than these generic arguments, an argument packet must be used. Status of the Card Services request is returned in the Status argument. Using functional notation, a generic Card Services call is as follows:

status = **CardServices(Service**, *Handle, Pointer, ArgLength, ArgPointer*)

All requests pass the service code of the request in the **Service** argument. Individual services and their service field values are described in later sections. Many requests require a Card Services handle

to identify some resource. These requests pass the handle in the *Handle* argument. Some requests require an additional pointer value which is passed in the *Pointer* argument.

Many Card Services requests have an additional argument packet which is pointed to by *ArgPointer*. The length of the argument packet is passed in the *ArgLength* argument. If the *ArgLength* argument is zero, there is no argument packet and the value of the *ArgPointer* argument is undefined.

The *ArgLength* argument may be used by a Card Services implementation to validate that the argument packet is appropriate for the indicated service. In different releases of this specification, the appropriate length of the argument packet may vary. Card Services uses this field to determine which packet length, and by extension, which version of the packet is being used by the client requesting the service.

See also ***Appendix-D, 9. Bindings*** for specific processor bindings for the generic Card Services arguments.

3.2.1.2 Argument Packet

Most argument packets are a fixed size determined by the particular service as implemented for a particular publication of this specification. Some argument packets can be variable in length. The size of these variable packets is determined by the caller. Variable length packets are used to contain data set by Card Services, for example, the Vendor Name ASCII string used to identify a particular version of Card Services.

The *ArgLength* argument indicates the length of the total packet. For variable length argument packets, there are additional fields in the packet that indicate the maximum length of the variable portion (set by the caller) and the actual length of the returned data (set by Card Services). Also some requests have more than one variable length argument. In this case, there is also an offset field that indicates where each additional variable length field begins.

The specific content of an argument packet is defined for each request that requires an argument packet.

3.2.1.3 Logical Sockets

The Card Services interface, except for **MapPhySocket**, uses logical sockets in identifying the socket a service is intended to address. The first physical socket on the first physical adapter is logical socket zero (0). The maximum logical socket is the total number of sockets present minus one.

3.2.1.4 Reserved Fields

Reserved fields and undefined bits shall be reset to zero before invoking a service because future releases of Card Services may define them. Future releases will use the reset value for behavior compliant with this release of Card Services.

Any reserved fields or undefined bits in fields returned by Card Services are reset to zero by Card Services so future releases of Card Services will be able to notify clients in a manner compliant with this release.

3.2.1.5 Multi-Byte Fields

All multi-byte fields are stored in binding specific format. Multi-byte data returned in bulk from a PC Card is kept in little-endian format with the least significant byte appearing first in memory. For

example, the **GetTupleData** and **Read/Write/CopyMemory** requests transfer the data without any byte swapping processing.

(See also ***Appendix-D, 9. Bindings.)***

3.2.1.6 Multiple Function PC Cards

Some PC Cards may contain multiple functions. To address a particular function on a PC Card, the client passes a card function number in the logical socket field of the appropriate request. Card Functions are numbered from zero to one less than the number of functions on the PC Card.

3.2.2 Presence Detection

The Card Services **GetCardServicesInfo** service is used to determine the presence of Card Services. If this request fails, Card Services is not present. If it succeeds, Card Services is present.

3.2.3 Initialization of Card Services

Card Services is designed to be implemented as an Operating System Dependent Device Driver or OS extension. If a processor supports different modes of operation, Card Services can assume that it is used in only a single mode. For example, processors in x86 architecture systems can run in Real Mode or Protect Mode. Card Services can assume that it is only used in one of these modes at any time.

During initialization, Card Services determines the state of the host environment. This includes determining available system memory, available I/O ports, IRQ assignments, installed PC Cards, and socket states.

Initialization is implementation specific.

After Card Services initializes, all Socket Services requests (080H through 0AEH) are blocked. Card Services returns an UNSUPPORTED_SERVICE error if any attempt is made to use these services. This prevents Socket Services clients who are unaware of the Card Services interface from crashing the system by making direct access to hardware through Socket Services. Such crashes could be caused by changing hardware state without Card Services being aware of the change. Should a Card Services aware client still require access to Socket Services, it may do so by using the entry point returned by the **ReturnSSEntry** service.

During initialization, Card Services determines all Socket Services implementations present so that it can manage the status change interrupt handling required for adapters. Socket Services status events are enabled based on client event masks. If no clients request an event, Card Services does not need to enable the event. Card Services records the event when it occurs and notifies any clients who have registered for its status change event and who have unmasked the event specified.

Card Services notifies registered clients and Memory Technology Drivers when events requiring callback notification have occurred. Notification is delayed until Card Services is in an enterable state which allows callback handlers registered with Card Services to make requests during event notification so they may reconfigure immediately to react to the event.

3.2.4 Return Codes

Card Services indicates success or failure of a request with the generic Status argument. If the Status argument is set to a non-zero value on return from a Card Services request, the request failed and the value in the Status argument describes why the request failed. If the Status argument is reset to zero

on return from a Card Services request, the request succeeded. (See ***Appendix-C, 8. Return Codes*** and see also ***Appendix-D, 9. Bindings.)***

3.3 Service Groups

The Card Services interface may be divided into five functional groups:

Client Services	provides for Client initialization and the callback registration of Clients.
Resource Management	provides basic access to available system resources, combining knowledge of the current status of system resources with the underlying Socket Services adapter control services.
Client Utilities	perform common tasks required by clients so that operations such as basic CIS tuple processing do not need to be duplicated in each of the client device drivers.
Bulk Memory Services	provides read, write, copy and erase memory services for use by file systems or other generic memory clients that want to be isolated from memory technology hardware details.
Advanced Client Services	provide specific services for clients with special needs.

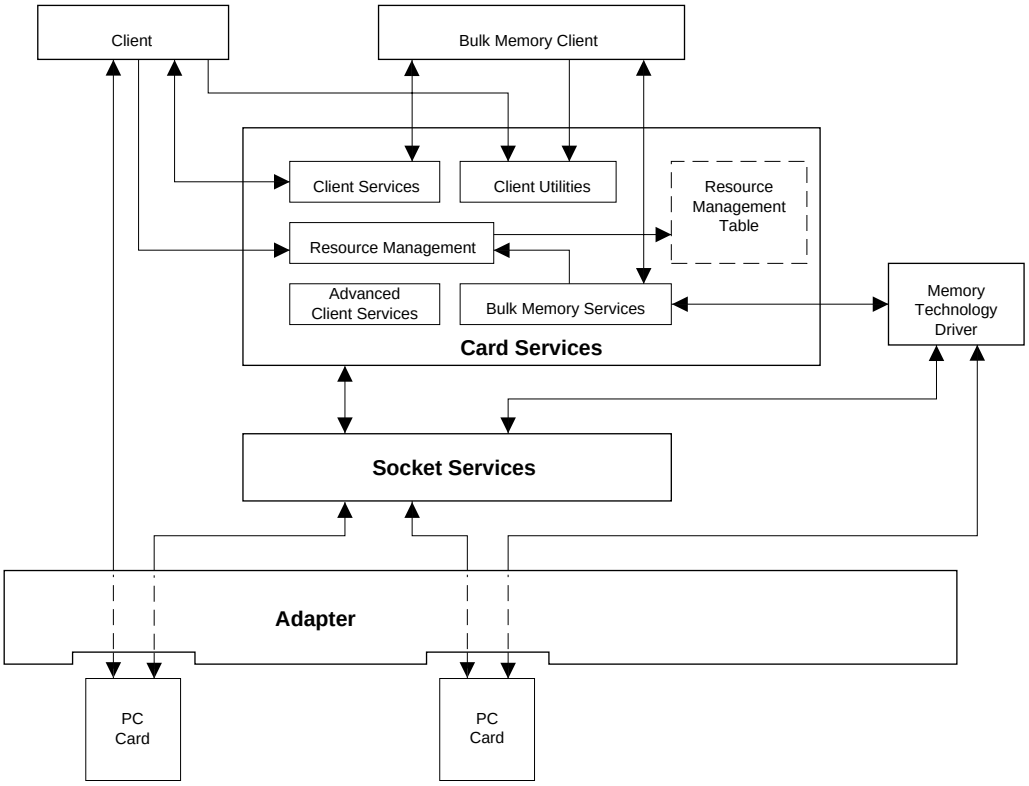

Figure 3-2: Card Services Diagram

3.3.1 Client Services

There are two types of services in the Client Services group:

- those that support client registration with Card Services to allow event notifications, and

- those that provide basic inquiry of PC Cards.

3.3.1.1 Client Registration

Card Services keeps track of the clients that can manipulate PC Cards with Client Services. The **GetCardServicesInfo** service is used by a client to determine the presence of Card Services. Clients invoke the **RegisterClient** service to inform Card Services of their presence and their interests in various events. The **DeregisterClient** service is used when a client is removing itself from the system.

A client specifies via **RegisterClient** whether it is a memory or I/O client device driver or a Memory Technology Driver. It does this by setting the appropriate bits in the Attributes field of the **RegisterClient** request.

Clients are notified of events of interest in an order dependent on the type of client and the order of registration. I/O clients are notified of PC Card insertion events first, and the last I/O client registered is notified first. This order allows the most up-to-date I/O client, which was the last to register, to configure the PC Card.

Memory Technology Drivers are notified next in the order of registration. This order allows more up-to-date MTDs to replace MTDs that previously registered for a region of a PC Card since all installed MTDs can register. Notifying MTDs after I/O clients allows the I/O client to configure the PC Card and interface before an MTD starts accessing memory regions. MTDs are assumed to be unaffected by the selected PC Card configuration.

Finally, memory clients are notified in the order of registration. This allows memory clients to make use of the PC Card in the appropriate configuration and also to make use of MTDs that are already in place to provide access to their memory areas.

Clients can use **RegisterClient** to be notified of events for all sockets in a system. **RequestSocketMask** can be used if a client only wants to be informed of events for a particular socket. **Get/SetEventMask** are used to change the event mask for the client.

When a client registers for callbacks, one of the fields in the argument packet provided is an event mask. This mask identifies the events that the client will be notified of by Card Services. The mask Card Services uses can be modified by a client via **SetEventMask** at any time to change the events of interest. When an event occurs, Card Services examines this mask. If enabled, Card Services notifies the client of this event. If not enabled, the client is not notified of the event.

3.3.1.2 Basic Card Support

The **ResetFunction** service resets the specified function of the PC Card in the specified socket. A hardware reset may cause the function to lose client-specific state. Before the hardware reset is performed, Card Services generates a **RESET_PHYSICAL** event. After the hardware reset, a **CARD_RESET** event is generated by Card Services. This allows clients to restore their specific function state. The **GetStatus** service returns information about the current status of a PC Card function and socket.

3.3.2 Resource Management

Card Services maintains a table of system resources usable by PC Cards and sockets. Resources are allocated to PC Cards via **RequestIO/IRQ/Window** services. These resources include I/O and memory address space and IRQs. For most efficient resource utilization, resources not needed permanently by a client may be returned to the resource pool by corresponding **ReleaseIO/IRQ/Window** services. For example, a memory window used to write to a PC Card's

configuration registers during card initialization can be returned to the resource pool after card initialization is complete.

The **ModifyWindow** and **MapMemPage** services allow a client to specify what portions of a PC Card's memory space are mapped into a dedicated memory window. These services also allow control of various attributes of accessing this memory, including access speed and memory space.

A client uses **RequestConfiguration** to configure a PC Card and socket for an I/O electrical interface and a selected configuration entry. **RequestIO/IRQ/Window** services must first be used to allocate any I/O and IRQ resources to the PC Card. After a suitable configuration is defined, **RequestConfiguration** is used to set the PC Card and Socket to the requested configuration. **ModifyConfiguration** can be used to make minor adjustments to a socket and PC Card configuration. The **ReleaseConfiguration** service reconfigures the PC Card and socket back to their initial memory only interface.

The **RequestSocketMask** service indicates that a client wishes to monitor the PC Card in a socket. This service allows the client to specify the events it is interested in monitoring. The **ReleaseSocketMask** service indicates that a client is no longer interested in socket event notifications.

In CardBus PC Cards, memory and I/O mappings are governed by Base Address Registers which are referenced when requesting memory and I/O resources for the cards. For CardBus PC Card mappings, the following additional information for each mapping is required:

- function number (0-7).
- Base Address Register number, a number from 1 to 7. Base Address Register 7 is always used for expansion ROM mappings. The combination of the socket number, function number and the Base Address Register number identifies the Base Address Register.

The following rule applies to requested mappings—the attributes of a given mapping indicate whether the mapping is a 16-bit PC Card or a CardBus PC Card mapping. There is no distinction made between the window handles resulting from the different kinds of mapping, or how the handles may be used by the Card Services client.

3.3.3 Bulk Memory Services

Bulk Memory Services provide services that can be used by clients such as file system utilities or XIP install utilities to avoid dealing with all the details of the various memory technologies that can be present on PC Cards. These services support a simple **Open/CloseMemory** and **Read/Write/CopyMemory** model of memory access. This model is similar to open, close, read, and write access to files in most operating systems.

Card Services determines PC Card memory regions during card insertion processing. Clients may determine areas in PC Card memory they wish to access by parsing the CIS or using the Card Services' services **GetFirst/NextTuple, GetFirst/NextPartition**, or **GetFirst/NextRegion**. Once a client determines the area of the PC Card they wish to access, they use the **OpenMemory** request and specify the absolute offset on the PC Card where the area begins.

OpenMemory returns a memory handle that is used for all subsequent read, write, copy and erase operations. These operations specify the location to be accessed relative to the start of the opened memory area. This allows clients to move data to and from PC Card memory as desired without concern as to where on the PC Card this particular memory area lies. A client performs a **CloseMemory** request to inform Card Services that it will no longer be accessing a memory area.

Performing an erase operation differs from the read, write and copy services. A client that needs to erase memory must register an erase queue with the **RegisterEraseQueue** request. The client then fills an erase queue entry identifying the socket and region of memory to erase. Next, the

CheckEraseQueue request is used to notify Card Services that one or more erase requests have been made in the erase queue. The actual erase operation is performed asynchronously. When the erase operation completes, the client is notified through the callback entry point provided in the erase queue header. In comparison, the read, write, and copy services return only after the requested action has been completed.

A client must use **DeregisterEraseQueue** to request Card Services to relinquish control of an erase queue. This must be invoked before a client is removed from memory. **DeregisterEraseQueue** can only be used when there are no queued erase requests in the erase queue.

Since erase operations return before the erase is complete, the client may be able to access other services while waiting for the erase completion. The physical construction of some cards (or memory components) may prevent some services until the erase operation in progress has been completed. In this event, the other requested operation is blocked (delayed) within Card Services until the erase is completed. Card Services does not notify the requester of erase completion until the blocked request is complete.

3.3.4 Client Utilities

Card Services clients may need to process a PC Card's Card Information Structure (CIS) to determine if and how they will interact with a card detected in a socket. (Some clients may receive all the information they require from the **CARD_INSERTION** event). The Client Utilities services reduce the code required for individual clients to perform such processing. **GetFirst/NextTuple** allow a client to traverse the CIS without being aware of how tuple links are evaluated. The client may concentrate on what to do with tuple data without having to duplicate the link traversal code. The client retrieves the contents of the tuple by using **GetTupleData**. Since many clients also require information describing regions and partitions derived from multiple tuples, **GetFirst/NextRegion** and **GetFirst/NextPartition** services are included to provide specific information without a client having any knowledge of the specific tuples containing the necessary data.

Clients should be aware that tuples may change between calls to any of the tuple processing services. Tuples could be changed by other clients (such as formatting utilities) that write tuples. **RequestExclusive** can be used to prevent other clients from accessing the PC Card during tuple modification.

3.3.5 Advanced Client Services

Advanced Client Services provides a miscellaneous set of services for use by client device drivers with special needs. **ReturnSSEntry** provides direct access to Socket Services. Clients that need direct Socket Services access can use this service to retrieve a reference to the location containing the Socket Services entry point. The **MapLogSocket/Window** and **MapPhySocket/Window** services have been provided to use the Socket Services interface on physical adapter hardware that has been allocated logically to a client by Card Services.

WARNING

Even though direct access to Socket Services is possible, such access may cause resource management services in Card Services to lose synchronization resulting in degraded operation, system crashes, or even hardware damage. Clients directly accessing Socket Services are responsible to ensure their usage does not interfere with Card Services.

Some advanced PC Card utilities may want to browse information present in Card Services to inform the end user of what is present in the host system. **GetFirst/NextClient** and **GetClientInfo** are provided to return information about clients registered with Card Services.

SetRegion can be used to define a memory region that an MTD can support when Card Services doesn't automatically recognize the region.

RegisterTimer allows a client to be called back after the specified delay. The actual callback is only performed when Card Services is enterable. This may result in a longer delay than specified.

Request/ReleaseExclusive allow a client to gain exclusive access to a PC Card. This could be used to allow a utility that writes or updates the CIS to have safe access to a PC Card.

ValidateCIS can be used to check the validity of the tuple chains in the CIS.

AddSocketServices allows additional driver versions of Socket Services to be added to an already initialized Card Services. **ReplaceSocketServices** allows a different and potentially newer version of Socket Services to replace an existing version. In addition, these services are used during dock-events by socket services handlers that are adding, changing or removing support for added or removed socket controllers.

3.4 Callback Interfaces

Card Services notifies clients of all events through a single callback interface. A client may be called back for any of the defined events. Except for **CARD_INSERTION** events, clients are not guaranteed to be notified in any particular order for a specific event. A client may be notified of any event at any time. Some events are sequenced in relation to other events. For example, Card Services notifies all clients of **RESET_REQUEST** before any client is notified of **RESET_PHYSICAL**. Specific event sequencing is defined in Section 3.5: Events.

Each event passes information to the client callback handler based on the type of event. Each type of event and the arguments it passes is discussed in the following sections. A client can make Card Services requests during callback processing.

Generic argument descriptions are used to define the information that is passed to the client's callback handler. Appendix H defines the processor specific arguments used. The generic arguments are:

Service	**Buffer**
Socket	**Misc**
Info	**Status**
MTDRequest	**ClientData**

Using functional notation, a Card Services callback is as follows:

```
Status = Callback(Service, Socket, Info, MTDRequest, Buffer, Misc,
         ClientData)
```

For all events, the **Service** argument contains a value identifying the event on entry to the client's callback handler. These event values are defined so that a single callback procedure can handle all types of events. The *ClientData* argument passes the information from the **RegisterClient** service's *ClientData* field.

The *Socket* argument identifies the socket and function affected by the event. For PC Cards with independently controllable functions the upper portion of the *Socket* argument is the function number (ranging from zero to one less than the number of functions on the card). The *Info* argument contains

other information specific to the event being reported. The *Status* argument is used by callback handlers to return information to Card Services. The *Buffer* argument is used to pass a pointer to a buffer for modification by the client. The *Misc* argument is used for miscellaneous information.

The *MTDRequest* is used specifically by MTDs to support read, write, copy, and erase requests.

A client event handler must preserve 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 all handlers have completed processing so it may perform any additional processing required.

Card Services sends separate notifications for each function on a multiple function PC Card.

3.4.1 Insertion

The **Service**, *Socket*, and *ClientData* arguments are passed to the client callback handler for a **CARD_INSERTION** event.

A client registers for insertion events with the **RegisterClient** service. Once registered, the client is notified each time a PC Card is inserted into a socket.

As part of the **RegisterClient** request, the client specifies interest in artificial insertion events for PC Cards inserted in sockets before the client registered. These artificial insertion events are intended to allow a client to establish its initial internal state without having to poll sockets to determine whether PC Cards are installed. Artificial insertion events are generated only once when a client first registers. Previously registered clients who have already completed initialization do not receive these artificial insertion events.

Since not all sockets may contain PC Cards and Card Services only sends artificial insertion events for occupied sockets, a client needs to determine when all such events have been generated. For this reason, Card Services generates a special **REGISTRATION_COMPLETE** event directly to the requester after all artificial insertion events have been sent.

3.4.2 Registration Completion

The **Service** and *ClientData* arguments are passed to the client callback handler for the **REGISTRATION_COMPLETE** event.

When a **RegisterClient** request is made, Card Services saves the client registration information and immediately returns to the client. Card Services then attempts to perform the registration in the background. When the registration processing is completed, Card Services notifies the requesting client's callback handler with a **REGISTRATION_COMPLETE** event.

3.4.3 Status Change

The **Service**, *Socket* and *ClientData* arguments are passed to the client callback handler for the following events:

BATTERY_LOW	**BATTERY_DEAD**
CARD_LOCK	**CARD_UNLOCK**
CARD_READY	**CARD_REMOVAL**
PM_SUSPEND	**PM_RESUME**
REQUEST_ATTENTION	**WRITE_PROTECT**

A client can receive events for any socket if it has enabled the event in its global event mask which is initially set by a client call to **RegisterClient**. The global event mask can also be set by **SetEventMask**.

To receive specific event notifications for the socket, a client can also perform an optional **RequestSocketMask** call before directly accessing a PC Card in a socket. A **RequestSocketMask** is useful since it allows a client to be notified of events for a specific socket. If **RequestSocketMask** is successful, the client receives events for the specified socket. Once installed, the client's callback handler is notified of status change events for the socket or PC Card installed in the socket. Clients may dynamically specify which status change events they are interested in by using the **SetEventMask** service.

If the PC Card in the socket experiencing the status change has multiple functions and the change is specific to a single function, the function number is placed in the upper half of the *Socket* argument.

Note: A separate event notification is sent to a client for each function on a PC Card.

3.4.4 Ejection/Insertion Requests

The **Service**, *Socket*, and *ClientData* arguments are passed to the client callback handler for the following events:

EJECTION_REQUEST	**EJECTION_COMPLETE**
INSERTION_REQUEST	**INSERTION_COMPLETE**

For the _REQUEST events, the *Status* argument must be set to SUCCESS on return to Card Services indicating that the client handled the request. If the *Status* argument is not set to SUCCESS, the request is rejected and the ejection or insertion will not be performed.

Note: The **EJECTION_REQUEST**, **EJECTION_COMPLETE**, **INSERTION_REQUEST**, and **INSERTION_COMPLETE** events refer to states related to driving a motor to insert or remove a PC Card and are not the same as the **CARD_INSERTION** or **CARD_REMOVAL** events described in other sections.

3.4.5 Exclusive

The **Service**, *Socket*, and *ClientData* arguments are passed to the client callback handler for the following events:

EXCLUSIVE_REQUEST	**EXCLUSIVE_COMPLETE**

When a **RequestExclusive** request is made to Card Services, it saves any information needed and immediately returns to the client. Card Services then attempts to make the PC Card available for exclusive use of the requesting client. For the **EXCLUSIVE** _REQUEST event, the *Status* argument must be set to SUCCESS on return to Card Services to indicate the client approves the request. If the *Status* argument is not set to SUCCESS, the request is rejected and the exclusive access will not be allowed.

When the exclusive processing is completed, Card Services notifies the requesting client at its callback entry with an **EXCLUSIVE_COMPLETE** event. If the **EXCLUSIVE_REQUEST** was rejected, **EXCLUSIVE_COMPLETE** is sent to the requesting client and the *Info* argument is set to the return code set by the client that rejected the request.

3.4.6 Reset

The **Service**, *Socket*, and *ClientData* arguments are passed to the client callback handler for the following events:

RESET_REQUEST	**RESET_PHYSICAL**
CARD_RESET	**RESET_COMPLETE**

When a **ResetFunction** request is made to Card Services, it notes that a reset has been requested and returns from the request. When the reset processing has been completed, Card Services notifies the client's callback handler with a **RESET_COMPLETE** event. If the **RESET_REQUEST** was rejected, **RESET_COMPLETE** is sent to the requesting client and the *Info* argument is set to the return code set by the client that rejected the request.

3.4.7 Client Information

The **Service** and *ClientData* arguments are passed to the client callback handler for the following events:

CLIENT_INFO

The *Buffer* argument is passed for the **CLIENT_INFO** callback and points to a data buffer to be filled with information by the client.

See the Service Reference for **GetClientInfo** for the format of the data buffer when the upper byte of the *Attributes* field is zero (0).

3.4.8 Erase Completion

The **ERASE_COMPLETE** event passes **Service**, *Socket*, and *ClientData* arguments to the client callback handler. The *Info* argument contains the erase queue entry number. The *Misc* argument contains the QueueHandle.

When an erase operation is requested of Card Services either via **RegisterEraseQueue** or **CheckEraseQueue**, Card Services only notes that there is new information in the erase queue and returns from the request. When an erase operation completes after having been processed in the background, the client callback handler specified in the erase queue header is notified of the **ERASE_COMPLETE** event.

3.4.9 MTD Request

The **MTD_REQUEST** event passes **Service**, *Buffer*, *MTDRequest*, *Socket* and *ClientData* arguments to the client callback handler. (See ***3.6 Memory Technology Drivers***.)

3.4.10 Timer

The **TIMER_EXPIRED** event passes **Service**, *Misc* and *ClientData* arguments to the client callback handler. The *Misc* argument contains the timer handle returned by **RegisterTimer**.

3.4.11 New or Removed Socket Services

The **SS_UPDATED** event passes the **Service**, *Socket*, *ClientData*, and *Info* arguments to the client callback handler. The *Socket* argument contains the logical socket number of the first socket supported by the newly installed Socket Services handler.

The *Info* argument is bit mapped as follows:

Bits 0 ·· 7	Number of sockets affected
Bit 8..9	New Sockets (bit mapped) 00 = PreviousReplaced 01 = SocketsAdded 10 = SocketsRemoved 11 = SocketRenumber
Bits 10 ·· 15	RESERVED (reset to zero)

The New Sockets field is bit mapped field where the value identifies what event has occurred. The value of zero (0) signals that the previously installed Socket Services handler was replaced. The value one (1) signals that additional sockets are present in the system and a new socket services handler is handling them. The value two (2) signals that sockets were removed from the system and that the socket services handler is no longer handling them. The value three (3) signals that the sockets are renumbering.

When the New Sockets field is three (3) then the *MISC* parameter will contain the new socket handle / number for the affected socket.

3.5 Events

This section describes the individual events that Card Services reports to clients. The following are discussed for each event:

- Specific cause(s) of the event,
- Pre-client processing by Card Services before notifying any clients,
- Expected client processing of the event, and
- Post-client processing by Card Services after all clients are notified.

Card Services sends separate notifications for each function on a multiple-function PC Card.

3.5.1 BATTERY_DEAD

Callback(BATTERY_DEAD, *Socket*, 0, *null*, *null*, 0, *ClientData*)

The **BATTERY_DEAD** event indicates the battery on a PC Card is no longer providing operational voltage.

Cause	The **BATTERY_DEAD** event occurs when the **BVD1** signal on a PC Card is negated. This signal may be available at the socket interface or in the pin replacement register. The negation of this signal results in a status change interrupt.
Pre-Client	Card Services notes a transition to a **BATTERY_DEAD** state. When the Card Services interface is available, Card Services notifies clients who have indicated their interest in **BATTERY_DEAD** events.
Client	A client processing **BATTERY_DEAD** notifications might warn the end-user that the PC Card is no longer capable of safely storing data if the PC Card is removed. How the client interacts with the end-user or what data loss preventive measures are taken is implementation specific.
Post-Client	Card Services does not perform any additional processing after notifying clients using the socket of the **BATTERY_DEAD** event.

Note: If the battery on a PC Card is dead when it is inserted, no **BATTERY_DEAD** event is generated. A **BATTERY_DEAD** event is only generated when the **BVD1** signal is negated after a PC Card has been inserted with the **BVD1** signal asserted.

See also **BATTERY_LOW**.

3.5.2 BATTERY_LOW

Callback(BATTERY_LOW, *Socket, 0, null, null, 0, ClientData*)

The **BATTERY_LOW** event indicates the battery on a PC Card is weak and is in need of replacement.

Cause	The **BATTERY_LOW** event occurs when the **BVD2** signal on a PC Card is negated. This signal may be available at the socket interface or in the pin replacement register. The negation of this signal results in a status change interrupt.
Pre-Client	Card Services notes a transition to a **BATTERY_LOW** state. When the Card Services interface is available, Card Services notifies clients who have indicated their interest in **BATTERY_LOW** events.
Client	A client processing **BATTERY_LOW** notifications might warn the end-user that the PC Card battery needs replacement. How the client interacts with the end-user or what data loss preventive measures are taken is implementation specific.
Post-Client	Card Services does not perform any additional processing after notifying clients using the socket of the **BATTERY_LOW** event.

Note: If the battery is weak on a PC Card when it is inserted, no **BATTERY_LOW** event is generated. A **BATTERY_LOW** event is only generated when the **BVD2** signal is negated after a PC Card has been inserted with the **BVD2** signal asserted.

See also **BATTERY_DEAD**.

3.5.3 CARD_INSERTION

Callback(CARD_INSERTION, *Socket, 0, null, null, ClientHandle, ClientData*)

The **CARD_INSERTION** event indicates a PC Card has been inserted in a socket or Card Services is creating artificial insertion events for PC Cards already in sockets. A separate **CARD_INSERTION** is generated for each function on a multiple function PC Card.

Cause

The **CARD_INSERTION** event occurs when the Card Detect pins (**CD1#** and **CD2#**) are asserted by the insertion of a PC Card. Card Services issues an **AcknowledgeInterrupt** request to Socket Services.

CARD_INSERTION events may also be artificially generated by Card Services after a new client performs a **RegisterClient** request. Artificial **CARD_INSERTION** events are only generated for sockets containing PC Cards. Registering clients may indicate whether they wish artificial **CARD_INSERTION** events for all PC Cards or only those without exclusive clients.

When an exclusive use of a PC Card is requested by **RequestExclusive**, a **CARD_INSERTION** event is generated to the requesting client if exclusive use can be granted. The **ReleaseExclusive** service also generates **CARD_INSERTION** events to all registered clients.

Pre-Client

If power was not previously applied to the socket Card Services enables **Vcc** to the socket. If the PC Card is not already in use, Card Services initiates a hardware reset of the PC Card. Sometime later, Card Services completes the reset.

Card Services then attempts to read the Card Information Structure (CIS).

Card Services uses device information from the Card Information Structure to create region description structures in its internal data area. Region description structures are created for all memory spaces on the PC Card. This information is later used by Card Services when clients request memory access. Other tuples processed by Card Services include the Function ID, Manufacturer ID, and bridge window requirements tuples.

Client

After the CIS is processed, all clients who have used the **RegisterClient** service to indicate their interest in insertion events are notified by Card Services. Clients may have enough information provided by a **GetConfigurationInfo** request to determine if they wish to use the PC Card. If they do not have enough information, clients may use Card Services to further process the CIS. Clients may process tuples directly using the memory **ReadMemory** services or use the tuple processing services **GetFirst/NextTuple**, **GetFirst/NextRegion** or **GetFirst/NextPartition**.

If a client wishes to be notified of events for only a particular socket, it uses the **RequestSocketMask** service which enables events for the specified socket. Clients may request exclusive use of a PC Card with the **RequestExclusive** service. If a previous client has requested the exclusive use of the PC Card, an exclusive request is rejected. The service **GetConfigurationInfo** can be used to determine whether the PC Card is currently in-use and if it is being exclusively used.

An I/O client can use **RequestConfiguration** to set the configuration desired for a PC Card and socket. Before an I/O client uses **RequestConfiguration** it must use the resource management services to allocate resources for the PC Card and/or socket as required. These services will not succeed if a previous client has already configured the PC Card and socket.

A memory client can simply use the memory access services to read, write, copy or erase data on the PC Card.

Post-Client

If no clients indicate they wish to use the socket with a successful **OpenMemory**, RequestExclusive, RequestWindow for a memory window, **RequestSocketMask**, or **RequestConfiguration** request after they have been notified of the **CARD_INSERTION** event, Card Services removes power from the socket.

See also **CARD_REMOVAL**.

3.5.4 CARD_LOCK

Callback(CARD_LOCK, *Socket, 0, null, null, 0, ClientData*)

The **CARD_LOCK** event indicates a mechanical latch has been manipulated preventing the removal of the PC Card from the socket.

Cause	Some sockets have hardware which can lock a PC Card into a socket to prevent inadvertent removal during operation. In addition, some sockets can report a change in the status of the locking hardware to warn that the card may be removed before it is actually removed. If a socket supports this capability, Card Services generates a **CARD_LOCK** event when the mechanical latch is locked.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client might respond to a **CARD_LOCK** event notification by setting internal state that it is safe to perform direct code execution from a PC Card. With the **CARD_LOCK** event, a client can be assured that directly executing code cannot be interrupted by the removal of a PC Card. Any client processing is implementation specific.
Post-Client	Card Services does not perform any post-client processing.

See also **CARD_UNLOCK**.

3.5.5 CARD_READY

Callback(CARD_READY, *Socket, 0, null, null, 0, ClientData*)

The **CARD_READY** event indicates a PC Card's **READY** line has transitioned from the busy to ready state.

Cause	A PC Card's **READY** line has been asserted.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client or MTD might respond to a **CARD_READY** event by completing an operation that is partially automated by a PC Card's on-board logic. It is expected that most clients will ignore **CARD_READY** events, performing polling to determine when PC Card's are in the ready state. Processing of this event is implementation specific.
Post-Client	Card Services does not perform any post-client processing.

Note: Most PC Card's negate **READY** whenever data is output to the card. As soon as the PC Card is ready to receive additional data, the **READY** line is asserted. For that reason, **CARD_READY** events may be extremely frequent. Clients may completely ignore such events and improve overall system response by resetting the **READY** bit in their global and socket event masks by using **SetEventMask** as necessary.

3.5.6 CARD_REMOVAL

Callback(CARD_REMOVAL, *Socket, 0, null, null, 0, ClientData*)

The **CARD_REMOVAL** event indicates a PC Card has been removed from a socket. A separate **CARD_REMOVAL** event is generated for each function of a multiple function PC Card.

Cause	The **CD1#** and **CD2#** pins in a socket are negated or a client requests a PC Card be reset. The **RequestExclusive** service generates **CARD_REMOVAL** events to clients that were registered for the socket. The **ReleaseExclusive** service also generates a **CARD_REMOVAL** event to the client exclusively using a PC Card after a **RequestExclusive**.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client must perform corresponding **Release_** resource requests for all successfully performed **Request_** resource services. **ReleaseConfiguration** must be requested first, followed by any additional **Release_** services. For example, if a client has routed the PC Card's **IREQ#** line with **RequestIRQ** then a **ReleaseIRQ** request should be made. This allows Card Services to update its internal database of system resource allocations and adjust socket hardware appropriately.
Post-Client	Card Services removes power from the socket.

Note: Clients should not attempt to make any further access to a socket after the **CARD_REMOVAL** event is received. Clients executing code directly from PC Card memory must pay particular attention to this event.

WARNING:

*Should a client fail to perform the appropriate **Release_** requests, Card Services' internal database of system resource allocations will not correctly reflect the resource state. Resources will be marked as in-use when they are in fact available.*

See also **CARD_INSERTION** and **CARD_LOCK**.

3.5.7 CARD_RESET

Callback(CARD_RESET, *Socket, ResetStatus, null, null, 0, ClientData*)

The **CARD_RESET** event indicates a hardware reset has occurred on a function of the PC Card in the specified socket.

Cause	A client requested a **ResetFunction** and the reset has been completed.
Pre-Client	Card Services has successfully performed a **RESET_REQUEST** notification and has physically reset the PC Card function.
Client	This is an opportunity for a client to re-establish any hardware state that existed before the PC Card function was reset. The *Info* argument contains SUCCESS if the reset has been successfully completed. If the reset was rejected, the *Info* argument contains IN_USE. If the reset was not successful, *Info* contains a return code indicating the reason for the failure. Handling of the event is implementation specific.
Post-Client	Card Services sends a **RESET_COMPLETE** notification directly to the client which requested the **ResetFunction** service.

See also **RESET_COMPLETE**, **RESET_PHYSICAL** and **RESET_REQUEST**.

3.5.8 CARD_UNLOCK

Callback(CARD_UNLOCK, *Socket, 0, null, null, 0, ClientData*)

The **CARD_UNLOCK** event indicates a mechanical latch has been manipulated allowing the removal of the PC Card from the socket.

Cause	Some sockets have hardware which can lock a PC Card into a socket to prevent inadvertent removal during operation. In addition, some sockets can report a change in the status of the locking hardware to warn that the card can be removed before it is actually removed. If a socket supports this capability, Card Services generates a **CARD_UNLOCK** event when the mechanical latch is unlocked.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client might respond to a **CARD_UNLOCK** event notification by setting internal state that it is not safe to perform direct code execution from a PC Card. Processing of the event is implementation specific.
Post-Client	Card Services does not perform any post-client processing.

See also **CARD_LOCK**.

3.5.9 CLIENT_INFO

Status = **Callback(CLIENT_INFO**, *0, 0, null, Buffer, 0, ClientData*)

The **CLIENT_INFO** event requests that the client return its client information data.

Cause	A requester used **GetClientInfo** to ask Card Services to return information about a client.
Pre-Client	Card Services calls the client for which information has been requested.
	Card Services may not pass the actual *ArgPacket* provided by the requesting client to the client specified by *ClientHandle* argument. Card Services may use an internal buffer for the **CLIENT_INFO** event notification. If Card Services does not pass the requesting client's actual *ArgPacket*, it copies all of the data in the *ArgPacket* into its internal buffer before sending it to the receiving client.
Client	The client shall copy its client information data into the buffer provided by Card Services.
Post-Client	Card Services returns the client information data to the requester.

WARNING

This is one of the events that require a response from the client's callback handler.

*Card services processes a **GetClientInfo** request to completion, without delay. The **CLIENT_INFO** event is transmitted to the target Client during this processing. The target Client is prohibited from using Card Services during the processing of the **CLIENT_INFO** event.*

See also **GetClientInfo**.

3.5.10 EJECTION_COMPLETE

Callback(EJECTION_COMPLETE, *Socket, 0, null, null, 0, ClientData*)

The **EJECTION_COMPLETE** event indicates a motor has completed ejecting a PC Card from a socket.

Cause	If a socket has hardware which can eject a PC Card from a socket, this event is generated when the ejection has been completed.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client might maintain an on-screen icon of the state of the socket. When this event is received, the icon could indicate ejection was complete and the PC Card could be removed from the socket. Processing of this even is implementation specific.
Post-Client	Card Services turns off the ejection motor if this is not performed automatically.

See also **EJECTION_REQUEST**.

3.5.11 EJECTION_REQUEST

```
Status = Callback(EJECTION_REQUEST, Socket, 0, null, null, 0,
         ClientData)
```

The **EJECTION_REQUEST** event indicates an end-user is requesting that a PC Card be ejected from a socket using a motor-driven mechanism.

Cause	If a socket has hardware which can eject a PC Card from a socket, this event is generated when an end-user requests ejection be performed.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client may ignore the event by returning with the *Status* argument set to SUCCESS. Card Services will then attempt to eject the PC Card from the socket. A client may also choose to prevent the ejection. In this case, the client should return from its callback handler with the *Status* argument not set to SUCCESS.
Post-Client	Card Services either starts the ejection motor after all clients are notified or ignores the ejection request depending on the state of the *Status* argument on return from the client's callback handler. After the ejection motor has completed ejecting the PC Card, the **EJECTION_COMPLETE** event is generated.

WARNING

This is one of the events that require a response from the client's callback handler.

See also **EJECTION_COMPLETE**, **INSERTION_COMPLETE** and **INSERTION_REQUEST**.

3.5.12 ERASE_COMPLETE

Callback(ERASE_COMPLETE, *Socket, EraseQueueEntryNum, null, null, EraseQueueHandle, ClientData*)

The **ERASE_COMPLETE** event indicates a queued erase request that is processed in the background has been completed. The client handle specified in the background erase queue data structure header identifies the client callback handler that is notified of this event.

Cause	The processing of a client's queued erase request has been completed by Card Services.
Pre-Client	Card Services performs the processing required by the client's queued erase request.
Client	The *EraseQueueHandle* is passed in the *Misc* argument. The *EraseQueueEntryNum* of the erase that was completed is passed in the *Info* argument. A client will check the EntryState for the affected erase queue entries to verify that the erase succeeded. The client then may immediately request that Card Services perform writes to record initial data structures in the newly erased block.
Post-Client	Card Services does not perform any post-client processing.

3.5.13 EXCLUSIVE_COMPLETE

Callback(EXCLUSIVE_COMPLETE, *Socket, ExclusiveStatus, null, null, 0, ClientData*)

The **EXCLUSIVE_COMPLETE** event indicates whether the client that requested exclusive access to a PC Card via the **RequestExclusive** service has received it.

Cause	A client uses **RequestExclusive** to gain exclusive access to a PC Card that may already be in use by other clients.
Pre-Client	Card Services has completed its processing. The *Info* argument indicates the client now has exclusive use if set to SUCCESS. If *Info* is not set to SUCCESS, the **RequestExclusive** failed and the client does not have exclusive access to the PC Card and the *Info* value (return code) indicates the reason for failure.
Client	If the request was successfully handled, the client can now use the PC Card exclusively.
Post-Client	Card Services does not perform any post-client processing.

See also **EXCLUSIVE_REQUEST**.

3.5.14 EXCLUSIVE_REQUEST

```
Status = Callback(EXCLUSIVE_REQUEST, Socket, 0, null, null, 0,
         ClientData)
```

The **EXCLUSIVE_REQUEST** event indicates that a client is trying to gain exclusive use of a PC Card via the **RequestExclusive** service.

Cause	A client uses **RequestExclusive** to gain exclusive access to a PC Card that may already be in use by other clients. Card Services sends **EXCLUSIVE_REQUEST** events to clients registered for the affected PC Card. The clients use the event return code to indicate whether or not they are willing to relinquish use of the PC Card.
Pre-Client	Card Services has already returned from the **RequestExclusive** service. This notification is being made from a background execution thread of Card Services. The Card Services interface is available.
Client	If the client is willing to relinquish its use of the PC Card, it returns with the *Status* argument set to SUCCESS. If the client is not willing to relinquish its use of the PC Card, the *Status* argument must not be set to SUCCESS.
Post-Client	If any client rejects the event, Card Services terminates notification processing and notifies the requesting client that the exclusive request failed. Once all clients have accepted the **EXCLUSIVE_REQUEST** event, Card Services sends **CARD_REMOVAL** events to all clients registered and then sends a **CARD_INSERTION** event to the requesting client. Finally, Card Services sends the **EXCLUSIVE_COMPLETE** event to the requesting client.

WARNING

This is one of the events that require a response from the client's callback handler.

See also **EXCLUSIVE_COMPLETE**.

3.5.15 INSERTION_COMPLETE

Callback(INSERTION_COMPLETE, *Socket, 0, null, null, 0, ClientData*)

The **INSERTION_COMPLETE** event indicates a motor has completed inserting a PC Card in a socket.

Cause	If a socket has hardware which can insert a PC Card into a socket, this event is generated when the insertion has been completed.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client might maintain an on-screen icon of the state of the socket. When this event is received, the icon could indicate insertion was complete and the PC Card was in the socket. This functionality is implementation dependent.
Post-Client	Card Services turns off the insertion motor, if this is not performed automatically.

See also **INSERTION_REQUEST**.

3.5.16 INSERTION_REQUEST

```
Status = Callback(INSERTION_REQUEST, Socket, 0, null, null, 0,
         ClientData)
```

The **INSERTION_REQUEST** event indicates an end-user is requesting that a PC Card be inserted into a socket using a motor-driven mechanism.

Cause	If a socket has hardware which can insert a PC Card into a socket, this event is generated when an end-user requests insertion be performed.
Pre-Client	Card Services does not perform any pre-client processing.
Client	A client may ignore the event by returning with the *Status* argument set to SUCCESS. Card Services will then attempt to insert the PC Card into the socket. A client may also choose to prevent the insertion. In this case, the client must return from its callback handler with the *Status* argument not set to SUCCESS.
Post-Client	Card Services starts the insertion motor after all clients are notified or ignores the request depending on the state of the *Status* argument on return from the client's callback handler. After the insertion motor has completed inserting the PC Card, the **INSERTION_COMPLETE** event is generated

WARNING

This is one of the events that require a response from the client's callback handler.

See also **EJECTION_COMPLETE**, **EJECTION_REQUEST** and **INSERTION_COMPLETE**.

3.5.17 PM_RESUME

Callback(PM_RESUME, *0, ResumeType, null, null, Mode, ClientData*)

The **PM_RESUME** event indicates that Card Services has received a resume notification from the host system's power management software. There are two types of resume events: NORMAL_RESUME (01H) and CRITICAL_RESUME (02H). The *ResumeType* is passed in the *Info* argument.

Card Services may or may not be notified before a host system enters suspend mode. Card Services uses a NORMAL_RESUME when a Card Services **PM_SUSPEND** event notification was sent to all power management aware clients before the system entered suspend mode. Card Services uses a CRITICAL_RESUME when a Card Services **PM_SUSPEND** event notification was not sent to clients before the system entered suspend mode. A system might enter suspend mode without notification when battery power is nearly depleted or the system goes directly to suspend mode when the end-user presses a button.

During resume processing, Card Services sends two **PM_RESUME** event notifications to each client interested in power management events. The first **PM_RESUME** event notification is sent with the *Misc* argument set to BEGIN_RESUME (01H). The second **PM_RESUME** event notification is sent with the *Misc* argument set to END_RESUME (02H).

Card Services sends a series of **CARD_REMOVAL** and **CARD_INSERTION** event notifications between the two **PM_RESUME** event notifications. If the state of a PC Card was changed while the system was suspended (for example, power was cycled on a socket), Card Services sends both **CARD_REMOVAL** and **CARD_INSERTION** event notifications.

If a PC Card was removed while the system was suspended, Card Services sends only a **CARD_REMOVAL** event notification. If a PC Card was inserted while the system was suspended, Card Services sends only a **CARD_INSERTION** event notification.

If the state of a PC Card was not changed during the suspend/resume cycle (see **PM_SUSPEND** - CONSERVE_POWER), Card Services only sends BEGIN_RESUME and END_RESUME event notifications. If a client has placed a PC Card in a reduced power state, the client must make sure the card is returned to a fully operational mode while processing an END_RESUME event notification.

All of the **CARD_REMOVAL** notifications for a socket are sent followed by all of the **CARD_INSERTION** notifications for the same socket. Then the next socket is processed. This event notification order allows Card Services and clients to re-synchronize their internal data state with PC Card and socket hardware state through normal card removal and insertion processing. It also minimizes the potential for resource conflicts during re-configuration since only the resources assigned to a particular PC Card are returned to the resource database at any one time.

Cause	Power Management software on the host system sends either a NORMAL_RESUME or CRITICAL_RESUME notification to Card Services.
Pre-Client **PM_RESUME** - BEGIN_RESUME	Card Services sends a **PM_RESUME** message with the *Info* Argument set to either NORMAL_RESUME or CRITICAL_RESUME and the *Misc* Argument set to BEGIN_RESUME to each power management-aware client (Clients indicate they are power management-aware by setting bit 8 of the *EventMask* during **RegisterClient**).
Client **PM_RESUME** - BEGIN_RESUME	The client sets an internal flag to note that the following **CARD_REMOVAL** and **CARD_INSERTION** messages are being generated by Card Services as part of resume processing.
Post-Client **PM_RESUME** - BEGIN_RESUME	Card Services sends a series of **CARD_REMOVAL** and **CARD_INSERTION** event notifications as described below for each socket in the host system.

Pre-Client **CARD_REMOVAL**	If a card was present in the socket before the system entered suspend mode and the state of a PC Card was changed while the system was suspended (for example, power was cycled on the socket), Card Services generates **CARD_REMOVAL** messages for the socket for each client interested in such messages. Card Services also sends a **CARD_REMOVAL** if a PC Card was removed while the system was suspended.
Client **CARD_REMOVAL**	If the client receiving this event notification configured the PC Card in the socket, the client releases the configuration and any allocated resources.
Post-Client **CARD_REMOVAL**	Card Services does not perform any post-client processing.
Pre-Client **CARD_INSERTION**	If a card was present in the socket before the system entered suspend mode and the state of a PC Card was changed while the system was suspended (for example, power was cycled on the socket), Card Services insures that the card is powered and properly reset. Card Services then generates **CARD_INSERTION** events for the socket for each client interested in such events . Card Services performs the same processing for PC Cards inserted while the system was suspended.
Client **CARD_INSERTION**	The client performs normal **CARD_INSERTION** processing. Power management aware clients may attempt to use the same resources for PC Card configuration and may additionally restore the PC Card to the card's state before the system entered suspend mode.
Post-Client **CARD_INSERTION**	Card Services does not perform any post-client processing.

Pre-Client **PM_RESUME** - END_RESUME	Card Services sends a **PM_RESUME** event with the *Misc* Argument set to END_RESUME to each power management aware client
Client **PM_RESUME** - END_RESUME	Reset internal state flag to indicate **CARD_INSERTION** and **CARD_REMOVAL** event notifications should again be processed normally. If the client set any PC Card to a reduced power state during a **PM_SUSPEND** - CONSERVE_POWER notification, the client must make sure the card is returned to a fully operational mode at this time.
Post-Client **PM_RESUME** - END_RESUME	Card Services returns from the resume notification.

See also **PM_SUSPEND**.

3.5.18 PM_SUSPEND

Status = **Callback(PM_SUSPEND**, *0, SuspendType, null, Buffer* | *null, 0, ClientData*)

The **PM_SUSPEND** event indicates that Card Services has received a suspend notification from the host system's power management software. There are four (4) types of suspend events. The suspend event type is passed in the *Info* argument. The suspend event types are:

QUERY_REQUEST (01H) - asks the client if it is OK to suspend the system. The client must respond immediately with either SUSPEND_OK (00H) or SUSPEND_NOT_OK (01H). The system shall send a CONSERVE_POWER or NO_POWER suspend event notification before actually entering any normal suspend mode. The client cannot take any action in response to this request that prevents the PC Card from being used normally. Regardless of the client's response to this request, the system may choose to suspend or choose not to suspend.

SNAPSHOT_REQUEST (00H) - asks the client if it is OK to suspend the system. If a client is handling a PC Card that has state information that would be lost if power is reduced or removed, the client must save the card's state information before returning from this request. Even though clients must reply with a SUSPEND_OK or SUSPEND_NOT_OK response, the system may choose to ignore the client's response. The client shall take no action in response to this request that prevents the PC Card from being used normally. No further notification of suspend activity is given if the system actually suspends. Regardless of the client response to this request, the system may choose to suspend or choose not to suspend.

CONSERVE_POWER (02H) - notifies the client that the system is suspending, but that PC Cards and sockets may remain powered. The *Buffer* argument points to a bit-mapped array representing the logical sockets supported by Card Services. The least significant bit in the least significant byte corresponds to logical socket zero (0). Card Services initializes all of the entries in the array to zero (0). Card Services removes power from a socket on completion of this notification if the bit in the array corresponding to the socket is reset to zero (0). If a client does not want power removed from the PC Card, the client must set the bit in the array corresponding to the socket to one (1). The client should attempt to reduce power consumption on all PC Cards it is using before returning from this notification, even if the client indicates Card Services may remove power. It is possible another client will request that Card Services leave the socket powered. A client must not reset a bit in the array to zero (0) if the bit is set to one (1) by another client.

Note: Even if a client indicates a PC Card should remain powered, the client should save any card state information that would be lost if power is removed. Power management software may elect to remove all power from the PC Card socket without further notification.

NO_POWER (03H) - notifies the client that the system is removing power from all PC Cards and sockets. For example, if a client is handling a PC Card that has state information that would be lost if power is removed, the client must save the card's state information before returning from this request. Card Services ignores any client response.

Note: Card Services does not send a **PM_SUSPEND** message for each socket managed by Card Services. A single **PM_SUSPEND** message of a given type is sent to Card Services clients for each equivalent type of Power Management notification received by Card Services.

Depending on the capabilities of the host system power management software, Card Services performs one of two processing sequences for **PM_SUSPEND** events. Which sequence Card Services uses depends on whether the host system power management software performs one or two pass

suspend notification. If the host uses a one pass notification, the only event notification the client receives is SNAPSHOT_REQUEST as described below.

Cause	Power Management software on the host system sends a suspend notification to Card Services.
Pre-Client	Card Services sends a SUSPEND_SNAPSHOT to all clients.
Client	A client saves any state information that would be lost if power is reduced or removed from any PC Card it is using. The client must not take any action that prevents the PC card from be used normally. The client returns either SUSPEND_OK or SUSPEND_NOT_OK.
Post-Client	Card Services returns SUSPEND_OK if all power management aware clients return SUSPEND_OK, otherwise Card Services returns SUSPEND_NOT_OK

If the host system power management software uses two pass suspend notification, Card Services first receives a query asking if the system should enter suspend mode. Before actually entering suspend mode, the host system power management software sends Card Services a second notification. The sequence for the first pass is:

Cause	Host system power management software sends a suspend query to Card Services.
Pre-Client	Card Services sends a QUERY_REQUEST event notification to power management aware clients.
Client	The client immediately responds to Card Services with either a SUSPEND_OK or SUSPEND_NOT_OK.
Post-Client	Card Services responds with SUSPEND_OK if all power management aware clients responded to the QUERY_REQUEST with SUSPEND_OK. Otherwise Card Services responds SUSPEND_NOT_OK.

When the host system power management decides to enter suspend mode (with or without a query pass), the following sequence is followed:

Cause	Host system power management software sends a suspend notification to Card Services.
Pre-Client	Card Services sends a CONSERVE_POWER or NO_POWER suspend event notification.
Client	The client shall record any PC Card state that may be lost when power is reduced or removed. If CONSERVE_POWER is received, the client may indicate the PC Card needs to remain powered by setting the bit in the array pointed to by the *Buffer* argument corresponding to the logical socket to one (1). In any case, the client shall place the PC card in a reduced power state, if one exists. If NO_POWER is received, the client should perform any actions required for an orderly shut-down.
Post-Client	For NO_POWER event notifications power is removed from the PC Card and socket. For CONSERVE_POWER event notifications, power is removed from all sockets whose corresponding bit in the array pointed to by the *Buffer* argument are zero (0).

WARNING:

*The Card Services interface may be busy during **PM_SUSPEND** event notifications. If a client requires the Card Services interface to perform **PM_SUSPEND** processing and finds the interface busy, the client should ignore the **PM_SUSPEND** notification and handle subsequent **PM_RESUME** notifications in the same manner as a CRITICAL_RESUME.*

See also **PM_RESUME**.

3.5.19 REGISTRATION_COMPLETE

Callback(REGISTRATION_COMPLETE, *0, 0, null, null, ClientHandle, ClientData*)

The **REGISTRATION_COMPLETE** event indicates a registration request that is processed in the background has been completed. The client handle specified in the **RegisterClient** request indicates the only client that will be notified of this event.

Cause	The processing of a client's **RegisterClient** request has been completed by Card Services.
Pre-Client	Card Services has completed notifying the client of any PC Cards that were already installed when the **RegisterClient** service was requested.
Client	Clients have probably been waiting for this event to signal they may continue their foreground processes that generated the original request. In this case, clients will most likely just set a semaphore indicating the request is complete. Then, when their foreground process again receives control, it will confirm the semaphore has been set and continue processing. This functionality is implementation specific.
Post-Client	Card Services does not perform any post-client processing.

3.5.20 REQUEST_ATTENTION

Callback(REQUEST_ATTENTION, *Socket, 0, null, null, 0, ClientData*)

The **REQUEST_ATTENTION** event indicates a PC Card is requesting attention from the host system.

Cause	A PC Card has set the REQ_ATN bit of the Extended Status Configuration register to one (1).
Pre-Client	Card Services does not perform any pre-client processing.
Client	The actions performed by a client are specific to the client and the PC Card. A PC Card could use this event to signal the card's client that an external source is requesting attention. For example, a FAX/data modem PC Card that is in a power down state could signal the client to return the PC card to an operational state to respond to an incoming telephone call.
Post-Client	Card Services does not perform any post-client processing.

3.5.21 RESET_COMPLETE

Callback(RESET_COMPLETE, *Socket, ResetStatus, null, null, 0, ClientData*)

The **RESET_COMPLETE** event indicates a **ResetFunction** request that is processed in the background has been completed. The client handle specified in the **ResetFunction** request identifies the client callback handler that is notified of this event. Other clients that may be using the card will not receive the **RESET_COMPLETE** event.

Cause	The processing of a client's **ResetFunction** request has been completed by Card Services.
Pre-Client	Card Services has completed the reset processing for the specified card.
Client	A Client has probably been waiting for this event to signal they may continue their foreground processes that generated the original request. In this case, clients will most likely just set a semaphore indicating the request is complete. Then, when their foreground process again receives control, it will confirm the semaphore has been set and continue processing. The *Info* argument contains SUCCESS if the reset has been successfully completed. If the reset was rejected, the *Info* argument contains IN_USE. If the reset was not successful, *Info* contains a return code indicating the reason for the failure. Processing of this event by the client is implementation specific.
Post-Client	Card Services does not perform any post-client processing.

3.5.22 RESET_PHYSICAL

Callback(RESET_PHYSICAL, *Socket, 0, null, null, 0, ClientData*)

The **RESET_PHYSICAL** event indicates a reset is about to occur on the specified function and socket.

Cause	A client requested a **ResetFunction** and no client rejected the previous **RESET_REQUEST** event.
Pre-Client	Card Services has successfully performed a **RESET_REQUEST** notification.
Client	This is an opportunity for a client to save any hardware state that may be lost when the PC Card is physically reset. Client processing is implementation specific.
Post-Client	Card Services sends a **CARD_RESET** to all clients and a **RESET_COMPLETE** notification directly to the client which requested the **ResetFunction** service.

See also **CARD_RESET**, **RESET_COMPLETE** and **RESET_REQUEST**.

3.5.23 RESET_REQUEST

Status = **Callback(RESET_REQUEST**, *Socket, 0, null, null, 0, ClientData*)

The **RESET_REQUEST** event indicates a physical reset has been requested by a client.

Cause	A client has requested a **ResetFunction** service.
Pre-Client	Card Services has already returned to the client requesting the **ResetFunction** service. This notification is being made from a background execution thread of Card Services. The Card Services interface is available.
Client	This is an opportunity for a client to prevent the reset request from occurring. The *Status* argument indicates whether the client will allow the request to complete. If the *Status* argument is set to SUCCESS on return from client notification, Card Services continues to notify other clients. If the *Status* argument is not set to SUCCESS on return from client notification, Card Services sends a **RESET_COMPLETE** event to the requesting client with the *Info* argument indicating the request was rejected.
Post-Client	Card Services sends a **RESET_PHYSICAL** notification and hardware reset is performed on the PC Card. Card Services then sends a **CARD_RESET**. Finally, Card Services sends a **RESET_COMPLETE** notification directly to the client which requested the **ResetFunction** service.

WARNING

This is one of the events that require a response from the client's callback handler.

See also **CARD_RESET**, **RESET_COMPLETE** and **RESET_PHYSICAL**.

3.5.24 SS_UPDATED

Callback(SS_UPDATED, *Socket, SSInfo, null, null, 0, ClientData*)

The **SS_UPDATED** event indicates that an **AddSocketServices** or **ReplaceSocketServices** request has changed the support provided for sockets.

Cause	**AddSocketServices** or **ReplaceSocketServices** has been called.
Pre-Client	Card Services uses the Socket Services handler to determine what hardware resources are now available, or have been removed, for the affected sockets.
Client	The client may react to the new, changed, or removed socket support. The client reaction depends upon the notification. For **SocketsRemoved** the client should perform any necessary internal data cleanup. For **SocketsAdded** the client should check if the socket(s) is occupied and whether the card is of interest (as if this were a CARD_INSERTION event). For a **SocketRenumber** notification the client should note the change of socket number for the affected socket.
Post-Client	No Post-Client processing is performed.

WARNING: SOME IMPLEMENTATIONS OF CARD SERVICES MAY BE UNABLE TO ACCEPT REQUESTS DURING THIS CALLBACK NOTIFICATION DUE TO RE-ENTRANCY LIMITATIONS CAUSING A BUSY STATE.

3.5.25 TIMER_EXPIRED

Callback(TIMER_EXPIRED, *0, 0, null, null, TimerHandle, ClientData*)

The **TIMER_EXPIRED** event indicates a timer registered by a client **RegisterTimer** request has expired. The *Misc* argument contains the timer handle returned by **RegisterTimer**.

Cause	The wait count has expired for a **RegisterTimer** request.
Pre-Client	Card Services has received a timer tick interrupt, noted the Card Services interface is available and the wait count is or will be decremented to zero (0) by this tick.
Client	The client may perform any processing it may have delayed.
Post-Client	No Post-Client processing is performed.

3.5.26 WRITE_PROTECT

Callback(WRITE_PROTECT, *Socket, WPState, null, null, 0, ClientData*)

The **WRITE_PROTECT** event indicates that the write protect status of the PC Card in the indicated socket has changed.

Cause	The write-protect switch on a PC Card has been moved.
Pre-Client	Card Services does not perform any pre-client processing.
Client	The client may check the value of the WPState field. If WPState is zero, the PC Card is not write-protected. If WPState is non-zero, the PC Card is now write-protected.
Post-Client	Card Services does not perform any post-client processing.

Note: Not all socket hardware is capable of reporting a change in a PC Card's write protect status. For this reason, a client should not rely on a **WRITE_PROTECT** notification as the sole method of determining a PC Card's write protect status.

3.6 Memory Technology Drivers

This section describes Memory Technology Drivers also known as MTDs. MTDs implement specific programming algorithms required to access memory devices on PC Cards. Card Services relies on MTDs to perform the actual read, write, copy, and erase services as well as internal card paging schemes necessary to access memory beyond the 64 Mbyte hardware signal limitation. Card Services uses the MTD Interface described in this section to access the MTDs which, in turn, use the MTD Helper Routines and the Media Access Table (MAT) Services to access the PC card. Additional information on the MTD Helper Services can be found in Appendix E.

3.6.1 Registration

MTDs register with Card Services like any other client by using the **RegisterClient** request. The data provided as an argument to **RegisterClient** includes an attribute field to indicate the requester is an MTD. The MTD's client callback handler is the entry point for MTD read, write, copy, and erase requests.

When an MTD is notified of **CARD_INSERTION** events, it can use the **GetFirst/NextRegion** services to determine if it wishes to handle read, write, copy and erase requests for memory on the PC Card. If an MTD elects to handle a region, it performs a **RegisterMTD** request to inform Card Services to use the MTD for all access to the region. If an MTD elects not to handle a region, the region is then handled by a previously installed MTD. By default, Card Services installs an MTD that supports read and write access to SRAM memory regions and non SRAM regions. In general, a client cannot depend on the state of memory after an erase request for a given MTD—the value read from a memory area that was just erased is undefined.

3.6.2 Card Services/MTD Interface

All requests to MTDs are made via the **MTD_REQUEST** event with the **Service**, *Socket*, *ClientData*, *Buffer* and *MTDRequest* arguments. The *MTDRequest* argument points to the following structure

Offset	Field	Size	Type	Value	Detail/Description
0	Length	2	I	N	Length of this packet
2	Socket	2	I	N	Logical Socket
4	SrcCardOffset	4	I	N	Source Card Offset for request
8	DestCardOffset	4	I	N	Destination Card Offset for request
12	TransferLength	4	I	N	Length of Request Read or Write = Bytes Erase = Power of Two
16	Service	1	I	N	MTD request service
17	Access Speed	1	I	N	Access Speed for Region
18	MTD ID	2	I	N	MTD's token from **RegisterMTD** for Region
20	MTDStatus	2	I/O	N	MTD Returned Status
22	Timeout Count	2	I/O	N	Timeout Count for Timer delayed requests
24	MAT	N	I	N	Media Access Table

When Card Services receives a read, write, or copy request from a client, it builds an *MTDRequest* packet and generates the **MTD_REQUEST** event to the appropriate MTD. Also, when Card Services finds an erase request in an erase queue, it constructs an erase request packet and passes it to the MTD for servicing.

Clients make read, write, copy, and erase requests using offsets relative to the beginning of the area identified in the **OpenMemory** request. MTDs require the absolute offset from the beginning of the PC Card. Card Services converts the offset address from relative to absolute in the MTD request packet before passing the request to the MTD.

Card Services performs some additional processing for a copy request. This processing varies depending on whether or not the adapter supports direct memory mapping.

If an adapter does not support direct memory mapping, Card Services converts a copy request into individual MTD read and write requests. An MTD does not receive a copy request in this case. Card Services breaks the request into a read followed by a write using an internal copy buffer. For example, if a copy request for 32 KBytes is made and the system only supports accessing through a 16 byte window, Card Services breaks the request into a number of 16 byte reads and writes to the MTD.

If an adapter supports direct memory mapping, Card Services maps the PC Card source area into a system address range and indicates this as the system buffer address, it sets the PC Card source offset, and requests an MTD copy operation. The PC Card source offset is provided to allow the MTD to manage power to the PC Card. This may be required to read from the PC Card.

If an address is passed to an MTD beyond the PC Card's 64 Mbyte hardware address signal limit, the MTD must point to the correct page within the PC Card. The MTD can change the card's page by modifying the address extension information in the appropriate Function Configuration Registers as described in the ***Electrical Specification***. Although the address extension provisions allow for a maximum of 2^{42} memory locations, this version of Card Services limits the maximum memory size to 4 GBytes.

SrcCardOffset contains the first memory location on the PC Card to read from for a Read request. For a copy request, it contains the memory offset on the PC Card that corresponds to the memory address passed in the *Buffer* argument. For a copy, this PC Card offset can be used by the MTD to appropriately manage access to the devices on the PC Card (if required).

DestCardOffset contains the first memory location on the PC Card of the destination for a Write and Copy request.

TransferLength contains the length in bytes of the request.

Service is a bit-mapped field defined as:

Bit 0 .. 1	Command: Erase (00H) Read (01H) Write (02H) Copy (03H)
Bit 2	DisableEraseBeforeWrite
Bit 3	VerifyAfterWrite
Bit 4	Ready Continued
Bit 5	Timeout Continued
Bit 6	Last in Sequence
Bit 7	First in Sequence

Command identifies the MTD service request.

VerifyAfterWrite and *DisableEraseBeforeWrite* are only valid for a Write command. *VerifyAfterWrite* requires the MTD to verify that the data was written correctly. *DisableEraseBeforeWrite* requires the MTD to not erase the memory before writing data. If this bit is reset to zero, the erase is only done for requests that are erase block aligned and a multiple of erase blocks. UNSUPPORTED_MODE is returned if an MTD doesn't support write verification.

Ready Continued indicates that this request was a continuation of a previous client request and is being continued by a **READY** event from the PC Card. *Timeout Continued* indicates that this request is being continued by a timeout. If neither of these bits are set to one, this request is an original request from a client. Both bits can be set to one if both events have occurred before Card Services was able to make the MTD request.

For a read, write, or copy *First in Sequence* and *Last in Sequence* indicate whether the request is part of a sequence that Card Services has broken up into smaller requests due to buffering or window allocation limitations. This information will typically be used to allow efficient power management of the PC Card. If this is the first request of the sequence, *First in Sequence* is set to one. If this is the last request, *Last in Sequence* is set to one. If a request corresponds to a single client request, both bits are set to one. If this request is neither the first nor last in the sequence, both bits are reset to zero.

Access Speed indicates the access speed for the memory being accessed. This field is defined the same way as the field for **GetFirst/NextPartition**.

MTD ID contains the value the MTD passed to Card Services in **RegisterMTD**. The MTD can use this value for its own purposes.

MTDStatus is set by the MTD when it returns a request with a SUCCESS or BUSY return code. This value is used by Card Services as the return code to the requesting client when SUCCESS is returned by the MTD. This field tells Card Services what event will trigger a retry of this request when BUSY is returned by the MTD.

Timeout Count indicates the timeout count when an MTD returns MTD_WAITTIMER or MTD_WAITRDY MTDStatus with a BUSY return code. A zero value causes the request to be retried at the next opportunity. The count is specified in 1 ms increments.

> *WARNING*
>
> *This timeout value along with the timeout granularity is not guaranteed and depends on system implementation details (see **RegisterTimer**).*

MAT contains the Media Access Table (NOT a pointer to the table) defined in a later section.

The MTD returns with the *Status* argument set to SUCCESS if the request has been completed. If an error occurred, the MTD places the error code in the MTDStatus argument. MTDs return the same codes to Card Services as are returned by Card Services to the client. If the MTD returns to Card Services with a return code of BUSY, some of the MTDStatus values inform Card Services of specific action that it should take for this MTD request.

These values determine the event that will trigger Card Services to retry the request.

MTDStatus	MTD State	Card Services reaction
MTD_WAITREQ (00H)	not currently able to service request	retries the request when MTD completes background operation
MTD_WAITTIMER (01H)	waiting for specified period before continuing request service	calls the MTD after the timeout period expires timeout count is specified in the *Timeout Count* field in the request packet before notifying MTD of timer expiration, *Timout Continued* is set to one in the Service field of the request packet
MTD_WAITRDY (02H)	waiting for **READY** before continuing request service	calls the MTD when the PC Card indicates **READY** the *Timeout Count* field is the maximum time Card Services will wait for **READY** the *Ready Continued* bit of the *Service* field is set to one to indicate a **READY** event continued request if the request times out, the *Timeout Continued* bit of the *Service* field is set to one
MTD_WAITPOWER(03H)	not currently able to service request due to lack of power	retries the request after a power change occurs that could affect this request

3.6.3 MTD Helper Interface

During the processing of a read, write, copy or erase request, MTDs can use the MTD Helper Services to control low level details of card access. The MTD Helper Services are all accessed via the entry point at the end of the MAT table included in the *MTDRequest* packet. MTDs use these services to perform socket and window management tasks. MTDs are NOT permitted to use the Card Services interface other than those provided by the Helper Service when processing read, write, copy, and erase requests.

The complete MTD Helper Service interface is described in Appendix E.

3.6.4 Erase Queuing

Card Services accepts additional erase requests while erase operations are in progress. MTDs may limit the number of simultaneous erases they can process. If an MTD cannot process an erase request when received because it has an erase in progress can not obtain sufficient programming current, it returns BUSY and MTD_WAITREQ or MTD_WAITPOWER to Card Services. Card Services leaves the erase request in the erase queue. Later, when the MTD notifies Card Services the erase in progress has been completed via a SUCCESS return code, any queued erase requests are re-attempted by Card Services.

3.6.5 Blocking

MTDs may not be able to satisfy read or write requests while an erase operation is in progress. MTDs handle this situation in the same manner as simultaneous erases. The MTD returns BUSY and MTD_WAITREQ. Unlike simultaneous erase, Card Services does not queue the request internally and return to the requesting client. In this case, Card Services blocks (delays) the request waiting for the erase in progress to complete. When notified by the MTD via a SUCCESS return code that the erase has completed, Card Services saves the erase completion status, and attempts the blocked request. Card Services notifies the client the blocked request has completed. Then, Card Services notifies the

requesting client that the erase causing the blockage has completed. If the erase request was not the blocking request, the erase request completion status is saved until the blocking request completes.

3.6.6 Card Services Request Retries

Card Services and MTDs cooperate to service client memory access requests. When an MTD returns to Card Services after processing a request, it indicates whether the request has been fully or partially processed. The code returned to Card Services helps determine what action Card Services takes next. If a request is fully processed, Card Services informs the requesting client. It does this directly or via a callback for erase requests. If the request is not fully processed, the MTD informs Card Services when to retry the request via the MTDStatus value in the *MTDRequest*, and Card Services saves the request on in an internal list.

The specific implementation details of Card Services are vendor dependent, but from an MTD perspective there are four logical lists of pending requests:

POWER	**TIMER**
MTD	**READY**

A power change triggers processing retries for all requests pending due to lack of power (the POWER list). A SUCCESS return code from an MTD triggers processing retries for all requests pending due to an WAIT_REQ (the MTD list). A timeout triggers a retry of the affected pending request (the TIMER list). A **READY** signal triggers processing retries for all requests awaiting a **READY** for the socket (the READY list).

Additionally, a request awaiting a **READY** can timeout. If the timeout happens for such a request only that request is retried. Finally, if there are requests awaiting a READY event and Card Services notices that the PC Card is **READY**, the pending READY requests are retried. Requests that were returned with a MTD_WAITPOWER or MTD_WAITREQ can also specify a timeout value to defer retry processing.

The Card Services response to specific MTD return codes and other events are described in the following table. An MTD will get retry requests from Card Services when the appropriate trigger events occur. An MTD must check whether it is possible to continue processing the retried request since the request may have been retried due to an unrelated event. If the request was not serviced, the MTD simply uses the appropriate return code to request that Card Services retry the request later.

For example, assume an MTD expects a READY event to allow further servicing for a request. Also assume there are other READY pending requests. Now when a **READY** is signaled from the PC Card, all pending READY requests for the socket will be retried by Card Services. The MTD must determine whether the READY event applies to each request presented by Card Services.

Card Services may not retry all pending requests at one time. Since MTD request retries are typically processed while in an interrupt handler, extensive processing may adversely affect host performance. Therefore, Card Services may process some requests and then wait for some period of time before continuing processing. However, Card Services ensures a single trigger event causes all requests waiting for that event to be retried. For example, Card Services ensures that all requests pending a READY event are processed after each **READY** is asserted.

Return Code	MTDStatus	Event	Card Services Response
BUSY	MTD_WAITPOWER		Put request on the power pending list
		Power change	Retry all power pending requests
BUSY	MTD_WAITTIMER		Put request on the timer pending list
		Timeout	Retry the timed-out request
BUSY	MTD_WAITRDY		Put request on the READY pending list for this socket
		READY	Retry all READY pending requests for the socket
		Timeout	Retry timed-out READY request
BUSY	MTD_WAITREQ		Put request on the MTD pending list
SUCCESS			Inform client of return code and, if erase, generate callback event, Retry all pending MTD requests for this MTD

3.6.7 Media Access Table

The Media Access Table is an array of pointers to the entry points for primitive routines that are used to access memory on a PC Card. The MAT is either built by using the Socket Services service **GetAccessOffsets** for register based sockets or is supported by Card Services itself for sockets with memory window mapping hardware.

The entry points are ordered as follows:

Service	Description
MATData	Pointer to the MAT data area.
CardSetAddress	Establishes access to a PC Card memory area.
CardSetAutoInc	Enables auto-incrementing addresses.
CardReadByte	Reads a byte from the memory area.
CardReadWord	Reads a word from the memory area.
CardReadWords	Reads words from the memory area. For use with AIMS PC Cards.
CardReadByteAI	Reads a byte from the memory area and automatically increments the memory address to the next byte.
CardReadWordAI	Reads a word from the memory area and automatically increments the memory address to the next word.
CardReadWordsAI	Reads a block of memory incrementing the memory address.
CardWriteByte	Writes a byte to the memory area.
CardWriteWord	Writes a word to the memory area.
CardWriteWords	Writes words to the memory area. For use with AIMS PC Cards.
CardWriteByteAI	Writes a byte to the memory area and automatically increments the memory address to the next byte.
CardWriteWordAI	Writes a word to the memory area and automatically increments the memory address to the next word.
CardWriteWordsAI	Writes a block of memory incrementing the memory address.
CardCompareByte	Compares a byte with a byte in PC Card memory.
CardCompareByteAI	Compares a byte incrementing the memory address.
CardCompareWords	Compares a block of memory against a block of PC Card memory.
CardCompareWordsAI	Compares a block of memory incrementing the memory address.
MTDHelperEntry	The entry point for MTD helper services of Card Services.

The definition of the Media Access Services are processor dependent and can be found in the Bindings section of Appendix F.

3.6.8 Virtual Memory Partitions/Regions

Some PC Cards have memory that cannot be directly accessed. For example, an Auto Indexing Mass Storage (AIMS) PC Card has control registers located in common memory that are used to access memory that is not directly addressable in common memory. Card Services defines such a memory area as a Virtual Region. Any partitions in such a memory area are defined as Virtual Partitions. A virtual region or partition is a memory area that is accessed by a client (via an MTD) by using memory addresses that aren't the same as its PC Card physical memory addresses.

MTDs can use special Card Services features to allow clients to use the **Open/CloseMemory** and **Read/Write/Copy/EraseMemory** requests to access such memory. **GetFirst/NextRegion/Partition** will return information about such memory areas to requesting clients.

For an MTD to provide access to a virtual region, it first uses **SetRegion** to inform Card Services of the existence of the region and its characteristics. This allows **GetFirst/NextRegion** requests to return this information to other clients. Next, the MTD uses **RegisterMTD** to inform Card Services that it supports access to this region.

A PC Card may have physical regions in addition to virtual regions. Virtual regions are not allowed to overlap their address ranges with any accessible physical regions. If a physical region would otherwise overlap its address ranges with a virtual region, the physical region must also be treated as a virtual region.

In order for an MTD to provide partition information about a partition in a virtual region, it may need to replace and simulate access to attribute memory tuples. This can be done by first relocating the attribute memory region as a virtual region to a different attribute memory address range. Then the MTD can create a simulated virtual attribute region that is located at the original physical attribute memory location. Whenever a client accesses attribute memory, the MTD can return the appropriate information.

3.6.9 Tuple Usage

Card Services performs automatic tuple processing in several cases relating to MTDs. The tuples used by Card Services are listed below along with the situations where they are processed.

Tuple	Event/Service	Usage
Device Information	**CARD_INSERTION** **GetConfigInfo** **GetFirst/NextTuple**	Determine access speed and size of region
JEDEC Identifier	**CARD_INSERTION** **GetConfigInfo** **GetFirst/NextTuple**	Determine JEDEC identifier
Function ID	**CARD_INSERTION** **GetConfigInfo** **GetFirst/NextTuple**	Determine function type, Determine system init mask
Manufacturer ID	**CARD_INSERTION** **GetConfigInfo** **GetFirst/NextTuple**	Determine manufacturer code, Determine manufacturer info
Format	**GetFirst/NextPartition**	Determine starting offset of partition, Determine partition size
Organization	**GetFirst/NextPartition**	Determine partition type
Device Geometry	**GetFirst/NextRegion**	Determine device characteristics.

During pre-client processing of **CARD_INSERTION** events, or during **GetConfigInfo** and **GetFirst/NextTuple** service events, Card Services identifies all of the regions present on a PC Card. This may require processing multiple Device Information and JEDEC Identifier tuples. (See the ***Metaformat Specification***.)

4. ASSUMPTIONS AND CONSTRAINTS

4.1 Auto Configuration of I/O Cards

Automatic configuration of I/O cards during a **CARD_INSERTION** event is implementation specific.

4.2 Compression

Card Services does not perform any compression or expansion of data presented to the read, write, or copy requests. Since Card Services has no knowledge of data structure or access patterns it is unable to perform on-the-fly compression. Compression is better performed by clients or operating systems.

4.3 EDC Generation

Card Services does not provide any support for Error Detection Code generation or validation. At present, there is no standard for how EDC should be handled in conjunction with stream I/O on a PC Card. Clients requiring EDC generation using socket or adapter-based hardware should make direct access to Socket Services.

4.4 BIOS or Device Driver

Card Services can be ROM-able. Card Services is intended to be an Operating System dependent loadable device driver or OS extension. During initialization, Card Services allocates its own RAM. Card Services may be dependent on specific details of the host system.

4.5 Interrupts Per Socket

The number of system interrupts available for routing PC Card **IREQ#** lines is implementation specific.

4.6 Mixed Media Memory Cards

Card Services should be implemented to support more than one type of memory on a PC Card. Card Services describes each homogeneous area of PC Card memory as a region. The number of regions supported per PC Card is implementation specific.

4.7 Multiple Partitioned Memory Cards

It is recommended that Card Services support more than one partition on a PC Card or within a region. The number of partitions supported per PC Card is implementation specific.

4.8 Use of Socket Services

Card Services makes all access to socket hardware through the Socket Services interface.

4.9 Interface Assumptions

4.9.1 Range Checking of Arguments

Card Services performs range checking only on items directly managed and numbered. For instance, Card Services checks that the specified logical socket number is valid or a PC Card is present in a socket being addressed. Card Services does not check that a specified card offset address is valid.

4.9.2 Configuration

How Card Services establishes its initial resource table describing the available system resources is implementation specific. The structure of the resource table is also implementation specific.

4.9.3 Abnormal Termination

Card Services does not perform any explicit processing if the operating system aborts a client process. It is the responsibility of the client to defend against unexpected termination and gracefully release any Card Services resources it may be using and deregister with Card Services.

4.9.4 Shared Data

The **GetFirst/Next** series of services share data between the initial **GetFirst** and subsequent invocations of **GetNext**. The **GetFirst** service invocation saves internal Card Services data in the argument packet, for use by subsequent **GetNext** invocations. The client is expected to preserve this internal Card Services Data for any subsequent use.

The validity of the shared *ClientHandle* parameter may be corrupted by any intervening invocation of **RegisterClient** or **DeregisterClient**.

4.10 Timeouts

Card Services does not perform any explicit timeouts. Card Services makes available limited timer services to MTDs to aid in monitoring the progress of background operations. If an MTD causes the blocking of a foreground operation, the MTD notifies Card Services when the blocking operation completes. MTDs are responsible for performing any required deadman timing.

5. SERVICE REFERENCE

The following sections describe the Card Services interface in detail. The services are listed alphabetically with their calling arguments identified for ease of reference. The functional notation used for a Card Services call is:

CardServices(**Service**, *Handle, Pointer, ArgLength, ArgPointer*)

For example:

CardServices(AddSocketServices, *null, SSEntry, ArgLength, ArgPointer*)

If an argument has a different value on call versus return, this is indicated by using a slash ("/") to separate the input versus output values. For example:

CardServices(GetConfigurationInfo, *null/ClientHandle, null, ArgLength, ArgPointer*)

If an argument is optional, this is indicated by using a logical or symbol ("|") to separate the defined parameters. For example:

CardServices(RequestIRQ, *ClientHandle, ISRAddress | null, ArgLength, ArgPointer*)

The argument name "*null*" is used for a pointer argument that is ignored for a particular request. Descriptive names are used to indicate the values required for other arguments. The name "*ArgLength*" indicates that the value for the correct size of the Argument Packet is used. The name "*ArgPointer*" indicates that a pointer to the Argument Packet is used.

The behavior, input and output parameters are described for each service. For ease of reference a parameter summary table is included for each service. For each service parameter this table specifies: offset in argument packet, name, size (in bytes), type, value, and a brief reference description.

The following abbreviations are used in the Card Services parameter summary tables. Parameter offset values in a parameter summary table are always expressed in decimal. Pointer values are in binding specific format. All other table values are expressed in hexadecimal unless stated otherwise.

Parameter Types		
Code	**Function**	**Description**
I	Input	Parameter written by the Client as an input to the service.
O	Output	Parameter returned by Card Services as an output of the service. This parameter is effectively read-only and can not be modified by a Client.
I/O	Input and Output	Parameter which requires an input value provided by the Client but which may have been modified by Card Services upon return from the service.

Parameter Values		
Code	**Meaning**	**Description**
XXXXH	Hexadecimal Data	Explicit hex value for the parameter.
ZERO	Zero	Zero value for the parameter.
N	Number	Variable data for a parameter.
BCD	Binary-Coded Decimal	BCD data for a parameter.

5.1 AccessConfigurationRegister (36H)

CardServices(**AccessConfigurationRegister**, *null, null, ArgLength, ArgPointer*)

This service allows a client to read or write a PC Card Configuration Register. This service must be used by Card Services clients to access a CardBus PC Card's registers in configuration space. For CardBus PC Cards these registers are referred to functionally, or by address. In other words, if the client wished to examine the first Base Address Register in function 2, the client would request Base Address Register 1 in function 2, or the DWORD at address 10H in configuration space.

This service is also used to access the four CardBus PC Card status registers associated with the function. The table below describes the arguments for this service.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket. The *Socket* field identifies the logical socket and the function for the PC Card to access. The least significant byte is the logical socket. The most significant byte is the function. Single function 16-bit PC Cards always use a zero (0) value for the function. CardBus PC Cards and multiple function 16-bit PC Cards use a value between zero (0) and one less than the number of functions on the PC Card
2	Action	1	I	N	The *Action* field may be set to READ (00H), WRITE (01H), READ_BYTE (02H), READ_WORD (04H), READ_DWORD (06H), WRITE_BYTE (03H), WRITE_WORD (05H), or WRITE_DWORD (07H). All other values in the *Action* field are reserved for future use. If the *Action* field is set to WRITE, WRITE_BYTE, WRITE_WORD, or WRITE_DWORD, the *Value* field is written to the specified Configuration Register. READ_BYTE (02H), READ_WORD (04H), READ_DWORD (06H), WRITE_BYTE (03H), WRITE_WORD (05H) and WRITE_DWORD (07H) can only be used if Offset field is set to FFH. Card Services does not read the Configuration Register after a write operation. For that reason, the *Value* field is only updated by a read request.
3	Offset	1	I	N	Byte offset to status register associated with the function indicated by the *Socket* field. This is relative to the PC Card configuration register base specified in **RequestConfiguration.** This must be a multiple of four for CardBus PC Cards. If the Offset is FFH, then the register to be read or written is a register in the CardBus PC Card function's configuration space.
4	Value	4	I/O	N	Value to read or to write. In the case where the register to be read or written is smaller than four bytes, e.g. for CardBus PC Card configuration space registers, the value in the least significant byte is taken first. For example, to set the *Bus Master* bit field in the *Command* register, the value 00000004H would be in this field.
8	Register	1	I	N	Which configuration space register is referred to. Ignored if *Offset* is anything other than FFH.

For registers in the CardBus PC Card function's configuration space, the *Register* field in the argument packet must be filled in. The allowable values for READ and WRITE actions are defined in the table below.

Register	Value	Corresponding Location in Configuration Space
BASE_ADDR_1	01H	First **Base Address Register**
BASE_ADDR_2	02H	Second **Base Address Register**
BASE_ADDR_3	03H	Third **Base Address Register**
BASE_ADDR_4	04H	Fourth **Base Address Register**
BASE_ADDR_5	05H	Fifth **Base Address Register**
BASE_ADDR_6	06H	Sixth **Base Address Register**
EXPANSION_ROM	07H	**Expansion ROM Base Register** at location 30H
BUS_MASTER	08H	Bit field 2 of the **Command** register at 04H
MEMORY_INVALIDATE	09H	Bit field 4 of the **Command** register at 04H
PALETTE_SNOOP	0AH	Bit field 5 of the **Command** register at 04H
BIST	0BH	**BIST** register at 0FH
CIS_POINTER	0CH	**CIS Pointer** register at 28H
LATENCY_TIMER	0DH	Latency Timer Register at 0DH

The allowable values for READ_BYTE and WRITE_BYTE actions (CardBus only) are defined in the table below.

Register	Value	Corresponding Location in Configuration Space
00H … FFH	00H ... FFH	Every configuration space register

The allowable values for READ_WORD and WRITE_WORD actions (CardBus only) are defined in the table below.

Register	Value	Corresponding Location in Configuration Space
00H, 02H, 04H … 0FEH	0000H … FFFFH	Every WORD aligned configuration space register

The allowable values for READ_DWORD and WRITE_DWORD actions (CardBus only) are defined in the table below.

Register	Value	Corresponding Location in Configuration Space
00H, 04H, 08H, 0FCH	00000000H … FFFFFFFFH	Every DWORD aligned configuration space register

A client must be very careful when writing to the 16-bit PC Card function *configuration* register at offset zero (0), the *Configuration Option* register. This has the potential to change the type of interrupt request generated by the PC Card or place the card in the reset state. Either request may have undefined results. The client should read the register to determine the appropriate setting for the interrupt mode (Bit 6) before writing the register.

If a client wants to reset a PC Card function, the **ResetFunction** service should be used. Unlike the **AccessConfigurationRegister** service, the **ResetFunction** service generates a series of event notifications to all clients using the function, so they can re-establish the appropriate card state after the reset operation is complete.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is invalid. Both 16-bit PC Card and CardBus PC Card argument lengths are legal.
BAD_ARGS	Specified arguments are invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
UNSUPPORTED_SERVICE	This service is not supported

See also the ***Electrical Specification***.

5.2 AddSocketServices (32H)

CardServices(AddSocketServices, *null, SSEntry, ArgLength, ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services entry point. Card Services calls Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Information about SS entry point
2	DataPointer	N	I	N	Pointer for SS Data Area (binding specific)

The *Attributes* field defines details about the new Socket Services entry point. The definition is binding specific. See the Bindings Section for specific implementations.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in a binding specific way. This field is defined the same as other (binding specific) pointers.

OUT_OF_RESOURCE is returned if Card Services cannot successfully manage this new Socket Services.

Note: A Card Services implementation may fail this request and return UNSUPPORTED_MODE if the provided pointers are for a processor mode unsupported by Card Services.

Return Codes

BAD_ARG_LENGTH	*ArgLength* value invalid - mode dependent
OUT_OF_RESOURCE	Out of internal RAM Space
UNSUPPORTED_MODE	Requested processor mode not supported

5.3 AdjustResourceInfo (35H)

CardServices(AdjustResourceInfo, *null/ClientHandle, null, ArgLength, ArgPointer*)

The Card Services internal database of system resources that are available for allocation to clients is managed by this service. The service is used to adjust or inquire about the system resource availability.

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	Add/Remove/Query resource
1	Resource	1	I	N	The resource type to adjust

The *Action* field has the following defined values:

0	RemoveManagedResource
1	AddManagedResource
2	GetFirstManagedResource
3	GetNextManagedResource
4 ·· 255	RESERVED

RemoveManagedResource is used to remove a system resource from the internal Card Services database. Once removed, the resource is no longer available for allocation by Card Services. It is assumed the resource is in use by some other entity in the host system and is not available to Card Services clients. IN_USE is returned if the resource, or any part of the resource is being used by a client when a **Remove** request is made.

AddManagedResource is used to add a system resource to the internal Card Services database. Once added, the resource is available for allocation to a requesting client. IN_USE is returned if the resource, or any part of the resource, is already being managed by Card Services.

GetFirst/NextManagedResource is used to return the current state of the internal Card Services database. These services are intended to be used by system utilities to display Card Services resource utilization. If a resource is currently allocated to a client, the returned *Attributes* field indicates the resource is allocated and the owning client's handle is returned in the *Handle* argument. If a resource is not currently allocated to a client, the returned *Handle* argument is undefined.

The *Resource* field identifies the system resource type:

0	Memory Range
1	I/O Range
2	IRQ
3	Reserved[1]
4	Socket Name
5 ·· 255	RESERVED

1. Legacy feature no longer supported.

The remainder of the argument packet is structured based on the system resource type.

For *Memory Range* resource types the argument packet has the following fields.

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	Add/Remove/Query resource
1	Resource	1	I	0	Memory Range resource
2	Attributes	2	I/O	N	Attributes of the memory range
4	Base	4	I/O	N	System_Base_Address
8	Size	4	I/O	N	Memory_Window_Size

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0 ·· 4	RESERVED (reset to zero)
Bit 5	Shared
Bit 6	Reserve for Specific Request
Bit 7	Allocated (set to one = true, output only)
Bit 8 ·· 15	RESERVED (reset to zero)

Shared is set to one if the memory range is being used but is sharable by other clients.

Reserve for Specific Request is only valid for **AddManagedResource** requests. It informs Card Services the memory range should only be assigned if it is specifically requested, or if there are no other resources that satisfy an ambiguous request. If the memory range is typically used by a standard PC peripheral, setting this bit can avoid having the range assigned to a client that doesn't care where its memory is located. This can improve the chances that the range will be available when a PC Card that needs the range is installed.

Allocated is only valid for **GetFirst/NextManagedResource** requests. It is set to one on return, if a client is currently using the Memory Range.

The *Base* field is the physical location in system address space where the memory range begins.

The *Size* field is the size of the memory range in bytes.

Return Codes (for memory adjustments)

BAD_ARG_LENGTH	*ArgLength* is not equal to twelve (12)
BAD_ATTRIBUTE	Specified attributes invalid
BAD_BASE	Starting system memory address is invalid
BAD_SIZE	Size of Memory Range is invalid
IN_USE	Memory Range, or part of the range, is already being managed
NO_MORE_ITEMS	There are no more Memory Ranges being managed by Card Services. Only valid for **GetFirst/Next** requests
OUT_OF_RESOURCE	No room in database to store updated information from **Add** or **Remove** request
UNSUPPORTED_SERVICE	This service is not supported

For I/O Range resource types there are two (2) possible argument packets which have exactly the same parameter definitions, but potentially a different size, to allow 32-bit addressing for I/O ranges. Both packets have the following fields:

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	**Add/Remove/Query** resource
1	Resource	1	I	1	I/O Range resource
2	Base Port	2 or 4	I/O	N	Base port address for range
4 or 6	Num Ports	1 or 4	I/O	N	Number of contiguous ports
5 or 10	Attributes	1	I/O	N	Bit-mapped
6 or 11	IOAddrLines	1	I/O	N	Number of I/O address lines decoded

Packet type 1, completely defined by the packet size indicated by sizeof(Base Port)=2 and sizeof(Num Ports)=1, is retained for backward compatibility purposes. Packet type 2, completely defined by the packet size indicated by sizeof(Base Port, Num Ports)=4, allow 32-bit addressing.

The *Base Port* field is the first port address of the I/O Range.

The *Num Ports* field is the number of contiguous ports in the I/O Range.

The *Attributes* field is bit-mapped. The following bits are defined:

Bit 0	Shared (set = true)
Bits 1 ·· 5	RESERVED (reset to zero)
Bit 6	Reserve for Specific Request
Bit 7	Allocated (set to one = true, output only)

Shared is set if the I/O Range is being used but is shareable by other clients.

Reserve for Specific Request is only valid for **AddManagedResource** requests. It informs Card Services the I/O Range should only be assigned if it is specifically requested, or if there are no other ranges that satisfy an ambiguous request. If the I/O Range is typically used by a standard PC peripheral, setting this bit can avoid having the range assigned to a client that doesn't care where its I/O ports are located. This can improve the chances that the I/O Range will be available when a PC Card that needs the range is installed.

Allocated is only valid for **GetFirst/NextManagedResource** requests. It is set to one on return, if a client is currently using the I/O Range.

The *IOAddrLines* field specifies the number of address lines decoded by the device using an I/O Range. If the device using an I/O Range does not decode all the I/O address lines used in the host system, Card Services needs to manage each I/O Range that has addresses in common with the number of address lines decoded. For example, if a device only decodes ten (10) address lines in a host system which uses sixteen (16) address lines, Card Services must manage all sixty-four (64) I/O Ranges that have common addresses in the lower ten (10) address lines. If such a device responds to an I/O range from 2F8H to 2FFH, Card Services must not allocate addresses ranges from 6F8H to 6FFH, EF8H to EFFH, etc.

Return Codes (for I/O Range resource types)

BAD_ARG_LENGTH	*ArgLength* is not equal to seven (7) or to (12)
BAD_ATTRIBUTE	Specified attributes invalid
BAD_BASE	Starting I/O address is invalid
BAD_SIZE	Size of I/O Range invalid
IN_USE	I/O Range or part of range is already being managed
NO_MORE_ITEMS	There are no more I/O Ranges being managed by Card Services. Only valid for **GetFirst/Next** requests.
OUT_OF_RESOURCE	No room in database to store updated information from **Add** or **Remove** request
UNSUPPORTED_SERVICE	This service is not supported

For IRQ Level resource types the argument packet has the following fields:

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	**Add/Remove/Query** resource
1	Resource	1	I	2	IRQ Level resource
2	Attributes	1	I/O	N	Bit-mapped
3	IRQ	1	I/O	N	IRQ Level

The *Attributes* field is bit-mapped. It specifies details about the specified IRQ.

The following bits are defined in the *Attributes* field:

Bit 0 ·· 1	IRQ type: 0 – Exclusive 1 – Time-Multiplexed Shared 2 – Dynamic Shared 3 – Reserved
Bit 2 ·· 5	RESERVED
Bit 6	Reserve for Specific Request
Bit 7	Allocated (set to one = TRUE, output only)

IRQ Type is set to ***Time-Multiplexed Shared*** if the IRQ is in use, but is shareable with other clients with only one client using the IRQ at a time. *IRQ Type* is set to *Dynamic Shared* if the IRQ is being used but is actively shareable.

Reserve for Specific Request is only valid for **AddManagedResource** requests. It informs Card Services the IRQ Level should only be assigned if it is specifically requested, or if there are no other levels that satisfy an ambiguous request. If the IRQ Level is typically used by a standard PC peripheral, setting this bit can avoid having the level assigned to a client that doesn't care what IRQ Level it uses. This can improve the chances that the IRQ Level will be available when a PC Card that needs the level is installed.

Allocated is only valid for **GetFirst/NextManagedResource** requests. It is set to one on return, if a client is currently using the IRQ Level.

The *IRQ* field is a binary value specifying the IRQ Level. It may range from zero (0) to a value one less than the number of IRQ Levels available in the host system.

Return Codes (for IRQ Level resource types)

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_ATTRIBUTE	Specified attributes invalid
BAD_IRQ	IRQ Level is invalid
IN_USE	IRQ Level is already being managed by Card Services
NO_MORE_ITEMS	There are no more IRQ Levels being managed by Card Services. Only valid for **GetFirst/Next** requests
OUT_OF_RESOURCE	No room in database to store updated information from **Add** or **Remove** request
UNSUPPORTED_SERVICE	This service is not supported

For the **Socket Name** resource type the argument packet has the following fields:

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	**Add/Remove/Query Resource**
1	Resource	1	I	4	Socket Name resource
2	Socket	2	I/O	N	Logical Socket Number
4	SocketNameOff	2	I/O	N	Offset to *SocketName* in argument packet
6	SocketNameLen	2	I/O	N	*SocketName* Length in characters (0=not named)
8	SocketNameCode	2	I/O	N	Socket Name Code 1:ISO/IEC 10646 USC-2 2:ASCII 3:JIS 4:S-JIS 5-FFFF:Reserved
N	SocketName	N	I/O	N	Socket Name (0 character terminated)

The *Socket* field is the logical socket number to which the socket name information applies. This is an input value for **Add/RemoveManagedResource** requests, and an output for **GetFirst/NextManagedResource** requests. The *SocketName* field is a string which identifies the physical socket in a way meaningful to the user. It is expected to be used by application and system software for display purposes. The string is encoded by *SocketNameCode*. The offset of the string from the beginning of the argument packet is specified in *SocketNameOff*. The actual length of the string (including the terminating 16-bit zero) is returned in the *SocketNameLen* field. The string should not include any formatting characters such as carriage returns, linefeeds, or tabs.

For **GetFirst/NextManagedResource** requests a *SocketNameLen* field returned with a value of zero (0) indicates that although the socket name service is supported, no name has been defined for the specified *Socket*. For **AddManagedResource** requests, setting the *SocketNameLen* field to zero (0) will force the socket name to be undefined.

Note that this call can succeed for **GetFirst/NextManagedResource** requests even if the argument packet is too small to accommodate the entire socket name string. If the buffer is not large enough to accommodate the name information, Card Services will return as much of the information as the *ArgLength* argument permits, even if only to indicate that the socket has not been named by setting the *SocketNameLen* field to zero. If the socket name is too long for the buffer, it will be truncated by Card Services on a valid 16-bit character boundary. In this case, the truncated string is still

guaranteed to be terminated by a 16-bit zero character, but the *SocketNameLen* will still reflect the actual length of the string, NOT the length of the truncated string.

For the processing of **RemoveManagedResource** requests, the *SocketName*, *SocketNameOff*, and *SocketNameLen* fields are not used, though they must be present to satisfy the minimum argument length.

Return Codes (for Socket Name resource types)

BAD_ARG_LENGTH	*ArgLength* is less than eight (8)
NO_MORE_ITEMS	There are no more Socket Names being managed by Card Services. Only valid for **GetFirst/Next** requests.
OUT_OF_RESOURCE	No room in database to store updated information from Add or Remove request.
UNSUPPORTED_SERVICE	This service is not supported

5.4 CheckEraseQueue (26H)

CardServices(CheckEraseQueue, *EraseQueueHandle, null, 0, null*)

This service notifies Card Services that the client has placed new entries into the queue to be serviced. Any erase requests contained in the erase queue should be initiated by Card Services. The *EraseQueueHandle* for the Erase Queue returned by **RegisterEraseQueue** is passed in the *Handle* argument.

Return Codes

BAD_HANDLE	Invalid erase queue handle

See also **RegisterEraseQueue** and **DeregisterEraseQueue**.

5.5 CloseMemory (00H)

CardServices(CloseMemory, *MemoryHandle/null, null, 0, null*)

This service closes an area of a memory card that was opened by a corresponding **OpenMemory**. Power may be removed from the socket if there are no other clients using the socket. The *MemoryHandle* returned by **OpenMemory** is passed in the *Handle* argument.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to zero (0)
BAD_HANDLE	Invalid memory area handle

See also **OpenMemory**, **ReadMemory**, **WriteMemory**, **CopyMemory**, and **CheckEraseQueue**.

5.6 ConfigureFunction (3CH)

CardServices(**ConfigureFunction**, *ClientHandle, ISRAddress/null, ArgLength, ArgPointer*)

This service configures a function in one step. By using the same structure that was created by the **InquireConfiguration** service, all resources for the function to be configured are requested in one **ConfigureFunction** call.

The *Request Type* field determines whether the client configuration is requesting or releasing the specified configuration. This field must be set to REQUEST to set the configuration and RELEASE to release the configuration. The Request Type field is defined as follows :

Bit 0	Requested Configuration: 0 - REQUEST 1 - RELEASE
Bits 1 ·· 7	Reserved (must be reset to zero)

If clients are configuring the card function to use an IRQ resource then the clients specify the address of a routine to handle interrupt events by providing a binding specific pointer to their routine in the *Pointer* argument as the *ISRAddress* parameter (otherwise the *Pointer* argument is null). Card Services installs a First-Level Interrupt Handler (FLIH) on the assigned interrupt vector that initially receives all interrupt notifications from the PC Card. Control is routed to Client handlers using a CALL instruction. On entry to the client handler the FLIH has preserved all registers and provided one hundred twenty-eight (128) words of stack space. A client routine requiring more stack space than this shall provide its own suitably sized stack space. When function specific interrupt processing is complete, the Client handler returns control to Card Services using a RET instruction. The handler shall indicate either that an interrupt condition was serviced by returning with the CARRY flag set or that the function did not require interrupt service by returning with the CARRY flag clear.

Please see **InquireConfiguration** service data structure definitions.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is invalid
BAD_HANDLE	*ClientHandle* is invalid, or on RELEASE *Request Type ClientHandle* does not match owning client or no configuration to release
BAD_TYPE	Requested Interface is not supported or requested Miscellaneous Feature setting is invalid
BAD_SOCKET	*Socket* or function is invalid
CONFIGURATION_LOCKED	PC Card function already configured
NO_CARD	No PC Card in socket
IN_USE	One or more resource requested for this configuration is not supported or is not currently available
UNSUPPORTED_SERVICE	This service is not supported

5.7 CopyMemory (01H)

CardServices(CopyMemory, *MemoryHandle, null, ArgLength, ArgPointer*)

This service reads data from a PC Card in the specified logical socket and writes it to another location in the same region. The *MemoryHandle* returned by **OpenMemory** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Source Offset	4	I	N	Card Source Address
4	Dest Offset	4	I	N	Card Destination Address
8	Count	4	I	N	Number of Bytes to transfer
12	Attributes	2	I	N	Bit-Mapped

The ***Source Offset*** is the relative starting location on the PC Card where the data to be copied originates. This offset is relative to the physical offset specified in the **OpenMemory** request that returned the *Memory Handle* used here.

The ***Dest Offset*** is the relative starting location on the PC Card where the data is to be placed. This offset is relative to the physical offset specified in the **OpenMemory** request that returned the *Memory Handle* used here.

Both offsets are relative to the physical offset specified in the **OpenMemory** request that returned the *Memory Handle.* For example, if an offset of 1000H was specified when the *Handle* was requested, a *Source Offset* of 500H would actually address physical offset 1500H in the PC Card's memory array.

The *Count* field is the number of bytes to copy. If the *Count* is zero, no transfer is made and the request returns successfully.

This service is not available for memory handles which address attribute memory. BAD_HANDLE is returned if such a request is made.

This service does not support overlapped copy requests. Attempting such a copy request results in undefined behavior.

The *Attributes* field is bit-mapped. The following bits are defined:

Bits 0 ·· 1	RESERVED (reset to zero)
Bit 2	DisableEraseBeforeWrite (set to one = true)
Bit 3	VerifyAfterWrite
Bits 4 ·· 15	RESERVED (reset to zero)

DisableEraseBeforeWrite is set to one to request that the memory area not be pre-erased before data is written to the PC Card. This erase is only done for requests that are erase block aligned and a multiple of erase blocks. *VerifyAfterWrite* is set to one to request that the data written be verified after the write. If an MTD doesn't support verification, Card Services may provide this support. **GetFirst/NextPartition/Region** can be used to determine the erase and verify capabilities of a memory area.

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to fourteen (14)
BAD_HANDLE	Invalid memory area handle
BAD_OFFSET	Invalid offset for source or destination
READ_FAILURE	Error reading from source
BAD_SIZE	Size of transfer is not valid
WRITE_FAILURE	Error writing to destination
NO_CARD	No PC Card in socket
WRITE_PROTECTED	Media is write-protected

See also **OpenMemory**, **ReadMemory**, **WriteMemory**, **CloseMemory**, and **CheckEraseQueue**.

5.8 DeregisterClient (02H)

CardServices(DeregisterClient, *ClientHandle/null, null, 0, null*)

This service removes a client from the list of registered clients maintained by Card Services. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument.

The client must have returned all requested resources before this service is called. If any resources have not been released, such as by any **Open** or **Request** call without a subsequent **Close** or **Release** call, IN_USE is returned.

If the client is an MTD, it is removed from handling access to any memory regions, i.e. the MTD had used **RegisterMTD** to support access to a region. Card Services notifies remaining MTDs via a **CARD_INSERTION** event for the affected sockets that the regions previously handled by this MTD need access support.

Note: Only MTDs are notified of these **CARD_INSERTION** events. Card Services first installs the default MTD for these regions so that if no notified MTD registers for a region, minimal access to the region is still available.

WARNING:
Clients should be prepared to receive callbacks until Card Services returns from this request successfully.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to zero (0)
BAD_HANDLE	Client handle is invalid
BUSY	MTD client has background task in progress
IN_USE	Resources not released by this client

See also **RegisterClient**.

5.9 DeregisterEraseQueue (25H)

CardServices(DeregisterEraseQueue, *EraseQueueHandle/null, null, 0, null*)

This service deregisters the erase queue that the client previously registered with Card Services.

DeregisterEraseQueue will fail if used to deregister an erase queue that has any pending erase entries. The *QueueHandle* returned by **RegisterEraseQueue** is passed in the *Handle* argument.

A return code of SUCCESS indicates the erase queue will no longer be serviced by Card Services.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to zero (0)
BAD_HANDLE	Invalid erase queue handle
BUSY	MTD client has background task in progress

See also **RegisterEraseQueue**.

5.10 GetCardServicesInfo (0BH)

CardServices(GetCardServicesInfo, *null, null, ArgLength, ArgPointer*)

This service returns the number of logical sockets installed and information about Card Services presence, vendor revision number, and release compliance information.

Offset	Field	Size	Type	Value	Detail/Description
0	InfoLen	2	O	N	Length of data returned by CS
2	Signature [0]	1	I/O	ZERO/'C'	ASCII 'C' Returned if CS installed
3	Signature [1]	1	I/O	ZERO/'S'	ASCII 'S' Returned if CS installed
4	Count	2	O	N	Number of Sockets
6	Revision	2	O	BCD	BCD Value of Vendor's CS Revision
8	CSLevel	2	O	BCD	BCD Value of CS Release
10	VStrOff	2	O	N	Offset to VendorString in argument packet
12	VStrLen	2	O	N	VendorString length (>=1)
14	FuncsPerSkt	2	O	N	Maximum number of functions per socket
N	VendorString	N	O	N	ASCIIZ vendor string buffer area

The *InfoLen* field returns the length of the Card Services Info that is valid in the argument packet on return. If *InfoLen* is greater than *ArgLength* argument then not all data fit in the supplied argument packet.

The *Signature* fields are returned as two ASCII characters, with *Signature[0***]** set to the ASCII character 'C' (43H) and *Signature[1]* set to the ASCII character 'S' (53H). The *Signature* fields should be reset to zero (0) before this service is invoked to prevent false sensing.

If Card Services is not present, the *Status* argument may contain the return code UNSUPPORTED_SERVICE. However, since Card Services may share its entry point with other service handlers, these other handlers may set the *Status* argument without a Card Services being present. In addition, if Card Services is not present, other service handlers may not even set the *Status* argument to indicate the service is unsupported. The *Signature* field should be checked for the ASCII characters 'CS' to confirm that a Card Services handler is present, if the *Status* argument is set to SUCCESS. If the *Status* argument is set to SUCCESS and the *Signature* field is set to the ASCII characters 'CS' on return, it may be assumed that Card Services is installed.

The *Count* field returns the number of logical sockets managed by Card Services. This value may be zero (0), if no sockets are present. Logical sockets are numbered from zero (0) to one less than the value returned in the Count field. Determining which physical adapter and socket correspond to a logical socket number may be done with the **MapLogSocket** request. The *Count* must include both CardBus PC Card and 16-bit PC Card-only sockets.

The *FuncsPerSkt* field returns the maximum number of functions managed by Card Services for each socket in the host system. This field was added to provide support for Multiple Function PC Cards. This field is not present in the argument packets returned by Card Services implementations that do not support Multiple Function PC Cards.

The *VendorString* field is an ASCIIZ string describing the Card Services implementer. It is expected to be used by system utilities for display purposes. The offset of the string from the beginning of the argument packet is specified in *VStrOff*. The actual length of the string (including the terminating zero) is returned in the *VStrLen* field. The string may include copyright legends and may be

formatted with carriage return and linefeed characters. If the *VStrLen* field is zero, the ASCIIZ string describing the implementer is not present. The *Revision* field is the vendor's internal revision number for this specific implementation of Card Services. It is stored as a BCD value with an implied decimal point (e.g. Revision 1.99 is 0199H.). The *CSLevel* field indicates the level of compliance with a Card Services publication. It is stored as a BCD value with an implied decimal point.

Publication	***CSLevel***
PC Card Standard Release 8.0 (April 2001)	0800H (8.00)
PC Card Standard Release 7.2 (November 2000)	0720H (7.20)
PC Card Standard Release 7.1 (March 2000)	0710H (7.10)
PC Card Standard Release 7.0 (February 1999)	0700H (7.00)
PC Card Standard Release 6.1 (April 1998)	0610H (6.10)
PC Card Standard Release 6.0 (March 1997)	0600H (6.00)
PC Card Standard Release 5.2 (May 1996)	0502H (5.02)
PC Card Standard Release 5.1 (November 1995)	0501H (5.01)
PC Card Standard Release 5.0 (February 1995)	0500H (5.00)
PCMCIA 2.1 / JEIDA 4.2	0210H (2.10)
PCMCIA 2.0 / JEIDA 4.1	0200H (2.00)

Return Codes

BAD_ARG_LENGTH	*ArgLength* is invalid
UNSUPPORTED_SERVICE	Card Services not installed

5.11 GetClientInfo (03H)

CardServices(GetClientInfo, *ClientHandle, null, ArgLength, ArgPointer*)

This service returns information describing a client. This information is expected to be used by browsing utilities. The *ClientHandle* returned by **RegisterClient** or **GetFirst/NextClient** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	MaxLen	2	I	N	Length of this packet
2	InfoLen	2	O	N	Length of Info returned by Client
4	Attributes	2	I/O	N	Bit-mapped (defined below)
6	ClientInfo	N	O	N	Client Information

The *InfoLen* field is the size of the *ClientInfo* argument packet the client needs for its ClientInfo data. If this value is greater than the *ArgLength* argument, then all of the client's data wasn't copied into the provided buffer. If this value is less than or equal to the *ArgLength* argument, then all of the client's data was copied into the provided buffer. *MaxLen* also contains the maximum length of the argument packet. This field is used by the client supplying the client info data. This field should have the same value as the *ArgLength* argument.

The *Attributes* field is bit-mapped. The field is defined as follows:

Bit 0	Memory client device driver (set = true)
Bit 1	Memory Technology Driver (set = true)
Bit 2	I/O client device driver (set = true)
Bit 3	**CARD_INSERTION** events for sharable PC Cards (set = true)
Bit 4	**CARD_INSERTION** events for cards being exclusively used (set = true)
Bits 5 ·· 7	RESERVED (reset to zero)
Bits 8 ·· 15	Info Subservice

The first five *Attributes* bits return the same information passed by the replying client to **RegisterClient** when it registered. The *Info SubService* field provides a mechanism for a requesting client to request other client specific information from the replying client. If *Info SubService* is zero, the *ClientInfo* field is structured as:

Offset	Field	Size	Type	Value	Detail/Description
0	MaxLen	2	I	N	Length of this packet
2	InfoLen	2	O	N	Length of Info returned by Client
4	Attributes	2	I/O	N	Bit-mapped (defined below)
6	Client Info	N	O	N	Client Information

6	Revision	2	O	BCD	BCD Value of Vendor client Revision
8	CSLevel	2	O	BCD	BCD Value of CS Release
10	RevDate	2	O	N	Revision Date
12	NameOff	2	O	N	Offset to ClientName String
14	NameLen	2	O	N	Length of ClientNameASCIIZ string
16	VStringOff	2	O	N	Offset in packet to VendorString buffer
18	VStringLen	2	O	N	Length of Vendor ASCIIZ string
N	NameString	N	O	N	Client Name ASCIIZ string
N	VendorString	N	O	N	Vendor String ASCIIZ string

The *Revision* field is the vendor's internal revision number for this specific implementation of the client. It is stored as a BCD value with an implied decimal point (e.g. Revision 2.03 would be 203H). The *CSLevel* field indicates the compliance level with a Card Services release number. It is stored as a BCD value with an implied decimal point. (e.g. Release 3.00 would be 300H). The *RevDate* field describes the revision date of the client implementation. It is stored packed in the same manner as dates in an MS-DOS directory entry. This format is:

Bits 0 ·· 4	Day. Ranges from 1 to 31
Bits 5 ·· 8	Month. Ranges from 1 to 12
Bits 9 ·· 15	Year. Relative to 1980. (1980 = 0, 1992 = 12, etc.)

The *NameLen* field is set by Card Services to the length required for the *NameString* field. The *NameString* field is the area Card Services will copy the ASCIIZ string describing the client. It is located at offset *NameOff* in from the beginning of the argument packet. It may be used by system utilities for display purposes. This string should NOT include carriage returns or linefeed characters. If *InfoLen* is greater than *MaxLen*, the entire client *NameString* may not have been copied into the *NameString* field. If *InfoLen* is less than or equal to *MaxLen*, the entire string was copied.

The *VStringOff* field is the offset from the beginning of the argument packet to the *VendorString* area that Card Services will copy the ASCIIZ string describing the client's implementer. The actual length required for the string is returned by Card Services in the *VStringLen* field. This string is expected to be used by system utilities for display purposes. It may include copyright legends and may be formatted with carriage return and linefeed characters. If *InfoLen* is greater than *MaxLen*, the entire client *VendorString* may not have been copied into the *VendorString* field. If *InfoLen* is less than or equal to *MaxLen*, the entire string was copied.

The *NameLen* and *VStringLen* fields can be zero to indicate that no strings are present.

If the *Info SubService* value is 80H - FFH the *Client Info* field is structured according to that client specific subservice. The subservice codes 01H - 7FH are reserved for future Card Services extensions. Card Services only ensures that the data returned by the replying client is returned to the requesting client and does not interpret or modify any such data.

Card Services may not pass the actual *ArgPacket* provided by the requesting client to the client specified by *ClientHandle* argument. Card Services may use an internal buffer for the **CLIENT_INFO** event notification. If Card Services does not pass the requesting client's actual *ArgPacket*, it copies all of the data in the *ArgPacket* into its internal buffer before sending it to the receiving client.

WARNING:

*Card services processes a **GetClientInfo** request to completion, without delay. The **CLIENT_INFO** event is transmitted to the target Client during this processing. The target Client is prohibited from using Card Services during the processing of the **CLIENT_INFO** event.*

Return Codes

BAD_ARG_LENGTH	*ArgLength* is less than six (6)
BAD_HANDLE	*ClientHandle* is invalid

5.12 GetConfigurationInfo (04H)

CardServices(GetConfigurationInfo, *null/ClientHandle, null, ArgLength, ArgPointer*)

This service returns information about the specified socket and PC Card configuration. The *ClientHandle* used to request this configuration (via **RequestConfiguration**) is returned in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	O	N	Bit-mapped
4	**VCC**	1	O	N	**VCC** Setting
5	**VPP1**	1	O	N	**VPP1** Setting
6	**VPP2**	1	O	N	**VPP2** Setting
7	IntType	1	O	N	Memory-only, Memory and I/O Interface, CardBus PC Card, or Custom Interface
8	ConfigBase	4	O	N	Card Base address of config registers
12	Status	1	O	N	Card Status register setting, if present
13	Pin	1	O	N	Card Pin register setting, if present
14	Copy	1	O	N	Card Socket/Copy register setting, if present
15	Option	1	O	N	Card Option register setting, if present
16	Present	1	O	N	Card Configuration registers present
17	FirstDevType	1	O	N	From Device ID Tuple
18	FuncCode	1	O	N	From Function ID Tuple
19	SysInitMask	1	O	N	From Function ID Tuple
20	ManufCode	2	O	N	From Manufacturer ID Tuple
22	ManufInfo	2	O	N	From Manufacturer ID Tuple
24	CardValues	1	O	N	Valid Card Register Values
25	AssignedIRQ	1	O	N	IRQ assigned to PC Card
26	IRQAttributes	2	O	N	Attributes for assigned IRQ
28	Base Port1	2	O	N	Base port address for range
30	Num Ports1	1	O	N	Number of contiguous ports
31	Attributes1	1	O	N	Bit-mapped
32	Base Port2	2	O	N	Base port address for range
34	Num Ports2	1	O	N	Number of contiguous ports
35	Attributes2	1	O	N	Bit-mapped
36	IOAddrLines	1	O	N	Number of I/O address lines decoded for a 16-bit PC Card. For CardBus PC Card this is ignored
37	Extended Status	1	O	N	Extended Status Register setting

38	Reserved[1]	1			
39	Reserved[1]	1			
40	NumIOWnds	1	O	N	Number of I/O windows in use on logical socket for this function
41	NumMemWnds	1	O	N	Number of memory windows in use on logical socket for this function
42	Custom Interface ID Number	4	O	N	Custom Interface ID Number (if IntType set to Custom Interface).

1. Legacy feature no longer supported.

This request returns information about the configuration of the specified socket and function on the PC Card in the socket. The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The fields from *Socket* to *IntType* are the same fields as for the **RequestConfiguration**. The *Attributes* field additionally has the following bits defined:

Bit 0	Exclusively Used (set = true)
Bit 2	CardBus PC Card Indicator (set = true)
Bit 8	Valid Client (set = true)

Exclusively Used is set to one when this PC Card is being exclusively used as requested by a successful **RequestExclusive** request.

If *CardBus PC Card Indicator* is set, then the card is a CardBus PC Card with a configuration space. If reset, then the card is a 16-bit PC Card.

If *Valid Client* is set to one, the client handle returned in the *Handle* argument is valid and configuration is in progress or locked. If *Valid Client* is reset to zero, the client handle returned in the *Handle* argument is not valid and configuration is not in progress or locked.

For CardBus PC Cards, **ConfigBase** has the same format as that of the *TPCC_RADR* field of the *CISTPL_CONFIG_CB* tuple.

The fields *Status* to *Option* and *Extended Status* are the values actually written to the corresponding 16-bit PC Card registers by Card Services during **RequestConfiguration** and may not reflect the current values in those registers. The other fields are the values that were passed to **RequestConfiguration**.

Note: The Option field has two parts. The lower six bits are the ConfigIndex field provided by the **RequestConfiguration** service. Bit 6 is determined by Card Services based on the interrupt type required by the client and the host hardware environment. Bit 7 is always reset to zero (0). This is the value actually written to the 16-bit PC Card Configuration Option Register by Card Services.

The *FirstDevType*, *FuncCode*, *SysInitMask*, *ManufCode*, and *ManufInfo* fields are the values from the corresponding tuples found in CIS on the PC Card. See the ***Metaformat Specification*** for tuple details. A value of 0FFFFH in any of the above tuple fields indicates that the corresponding tuple is not in the CIS on the PC Card.

The *CardValues* field indicates which of the Card Configuration register values were written to the PC Card. If the PC Card was configured during POST by the BIOS, Card Services may not know what

values were written to the PC Card registers. For CardBus PC Cards, this field does not apply and will always be reset to zero. The field is bit-mapped as follows:

Bit 0	Option Value Valid
Bit 1	Status Value Valid
Bit 2	Pin Replacement Value Valid
Bit 3	Copy Value Valid
Bit 4	Extended Status Value Valid
Bits 5 ·· 7	RESERVED (Reset to zero)

The *AssignedIRQ* and *IRQAttributes* fields are the same as defined in **RequestIRQ**. If the socket is not configured to use an IRQ level (**RequestIRQ** has not been successfully invoked), the *AssignedIRQ* field is set to FFH.

The *Custom Interface ID Number* field is used when the IF_CUSTOM interface type is set in the *IntType* field. This Interface Number is a PCMCIA and JEITA jointly assigned value that identifies a specific custom interface. (See also the discussion of Custom Interface Subtuples under CISTPL_CONFIG in the ***Metaformat Specification***.)

When the packet length is 37 and two or less I/O ranges have been allocated and the I/O ranges can be described using 16-bit fields, the *Base Port1, Num Ports1, Base Port2* and *Num Ports2, and IOAddrLines* fields will describe *the range(s). When more than two I/O ranges have been allocated or the I/O ranges require more than 16-bit* description fields, the *Num Ports1* and *Num Ports2* fields will return zero (0).

If the packet length of 42 is used then the number of windows in use on a logical socket for the specified PC Card function is indicated by the NumIOWnds and NumMemWnds fields and the fields from *Base Ports1* to *IOAddrLines* are ignored. When the *NumIOWnds* and *NumMemWnds* fields are used, the actual window characteristics are enumerated using the **GetFirst/NextWindow** services.

If the packet length of 46 is used then the Custom Interface ID Number field is returned in addition to the data returned when a packet length of 42 is used. The Custom Interface ID Number is not returned when packet lengths of 37 or 42 are used.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirty-seven (37), forty-two (42), or forty-six (46).
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_CARD	No PC Card in socket

5.13 GetEventMask (2EH)

CardServices(GetEventMask, *ClientHandle, null, ArgLength, ArgPointer*)

This service returns the event mask for the client. The *ClientHandle* returned by **RegisterClient** or **GetFirst/NextClient** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined below)
2	EventMask	2	O	N	Bit-mapped (defined below)
4	Socket	2	I	N	Logical socket

The *Attributes* field is bit-mapped. It identifies the type of event mask to be returned. The field is defined as follows:

Bit 0	Event mask for socket and function indicated
Bits 1 ·· 15	RESERVED (Reset to zero)

If Bit 0 is reset, the global event mask is returned. If Bit 0 is set, the event mask for this socket and function is returned. **RequestSocketMask** must have been requested by this client before the socket event mask can be returned. BAD_SOCKET is returned if the client has not specifically registered for this socket.

The *Event Mask* field is bit-mapped. Card Services performs event notification based on this field. The low-order eight bits specify events noted by Socket Services. The upper eight bits specify events generated by Card Services. The field is defined as follows:

Bit 0	Write Protect
Bit 1	Card Lock Change
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Dead
Bit 5	Battery Low
Bit 6	Ready Change
Bit 7	Card Detect Change
Bit 8	PM Change
Bit 9	Reset events
Bit 10	SS Update
Bit 11	Request Attention Change
Bits 12 ·· 15	RESERVED (Reset to zero)

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

See the Insertion callback section for additional information about handling events.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to six (6)
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_CARD	No PC Card in socket

5.14 GetFirstClient (0EH)

CardServices(GetFirstClient, *null/ClientHandle, null, ArgLength, ArgPointer*)

This service returns the first *ClientHandle* of the clients that have registered with Card Services. The *ClientHandle* is returned in the *Handle* argument. The *Status* argument is set to NO_MORE_ITEMS, if there are no registered clients.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)

When the *Attributes* Bit 0 is set (1), the *Socket* field is used to qualify the *ClientHandle*s considered. The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The bits are defined as follows:

Bit 0	0 = consider all registered clients 1 = consider clients for this socket only
Bits 1 ·· 15	RESERVED (Reset to zero)

If Bit 0 is set to one (1), only the clients accepting events for this socket and function are returned. If Bit 0 is reset to zero (0), all clients registered with Card Services are returned.

WARNING

*If another client performs a successful **RegisterClient** or **DeregisterClient** request between a **GetFirstClient** and **GetNextClient** or between any two **GetNextClient** requests, the results are not predictable.*

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_MORE_ITEMS	No clients are registered
NO_CARD	No PC Card in socket

5.15 GetFirstPartition (05H)

CardServices(GetFirstPartition, *null, null, ArgLength, ArgPointer*)

This service returns device information for the first partition on the card in the specified socket based on the PC Card's CIS. If there are no partitions, the *Status* argument is set to NO_MORE_ITEMS.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Partition Attributes Field
4	TupleMask	1	O	N	Bit-mapped (defined below)
5	Access Speed	1	O	N	Window Speed Field
6	Flags	2	O	N	CS Partition Flags Data
8	Link Offset	4	O	N	CS Partition Link Data
12	CIS Offset	4	O	N	CS Partition CIS Data
16	Card Offset	4	O	N	Card Memory Region Offset
20	Part Size	4	O	N	Partition Size
24	EffBlockSize	4	O	N	Erase Block Size
28	PartMultiple	2	O	N	Partition Multiple (Erase Block units)
30	JEDEC ID	2	O	N	Partition JEDEC Memory ID Code
32	PartType	2	O	N	Partition Type Field

For 16-bit PC Cards, the *Socket* field describes the logical socket containing the desired card.

For CardBus PC Cards, the *Socket* field identifies the logical socket and function. The least significant byte is the logical socket. The most significant byte of the *Socket* field is the function. Allowable functions are numbered from 0 to 7.

The *Attributes* field is bit-mapped. The bits are defined as follows:

Bit 0	Memory type (set = attribute, reset = common)
Bit 1 ··2	RESERVED (Reset to zero)
Bit 3 ··4	Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use
Bits 5 ·· 7	RESERVED (Reset to zero)
Bit 8	Virtual Partition (set to one = true)
Bits 9 ·· 10	Write/Erase interactions: 0 - Write without Erase 1 - Write with Erase 2 - Reserved 3 - Write with Disable-able Erase
Bit 11	Write with Verify
Bit 12	Erase Requests Supported
Bits 13 ·· 15	Base Address Register number (1-7).

For CardBus PC Cards *Memory Type* will always be reset.

Prefetchable / Cacheable applies to CardBus PC Cards only and is set by the service. 16-bit PC Cards shall ignore this field.

Virtual Partition is set to one when the partition can only be accessed via an appropriate MTD, i.e. the partition is not addressable simply by presenting addresses to the PC Card (e.g. via a memory window).

Write without Erase indicates no erase is done before a write. *Write with Erase* indicates writes that are erase block aligned and multiple erase block sized are erased before being written. *Write with Disableable Erase* indicates the *WriteMemory* attribute *DisableEraseBeforeWrite* can be used to control if an erase before write is not done. *Write with Verify* is set to one if writes can be verified after writing. The *WriteMemory* attribute *Verify* is used to request a verified write. *Erase Requests Supported* indicates that erase requests via an *EraseQueue* are supported for this partition.

The *Base Address Register* number indicates the associated Base Address Register on the CardBus PC Card. Base Address Register number seven (7) always refers to the Expansion ROM Base Address Register.

The *Tuple Mask* field is bit-mapped. Some file systems which use the entire common space on PC Cards and do not have writable attribute space do not have partition-related tuples in the Card Information Structure. Card Services may be able to recognize partition information without definition in tuples. This field indicates whether partition information was derived from tuple information or whether Card Services determined returned values empirically. The following bits are defined:

Bit 0	Access Speed from tuples (set = true)
Bit 1	Card Offset from tuples (set = true)
Bit 2	Part Size from tuples (set = true)
Bit 3	EffBlockSize from tuples (set = true)
Bit 4	Part Multiple from tuples (set = true)
Bit 5	JEDEC ID from tuples (set = true)
Bit 6	Part Type from tuples (set = true)
Bit 7	Reserved (reset to zero 0).

The *Access Speed* field is bit-mapped as follows:

Bits 0 ·· 2	Device speed code, if speed mantissa is zero Speed exponent, if speed mantissa is not zero
Bits 3 ·· 6	Speed mantissa
Bit 7	Wait (set = use **WAIT#**, if available)

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero, the lower bits are a binary code representing a speed from the following table:

Code	Speed
0	(Reserved - do not use)
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5 .. 7	(Reserved - do not use)

The *Flags*, *Link Offset* and *CIS Offset* fields in the argument packet description are for internal use by Card Services. Card Services initializes them to the appropriate values. The client must preserve these values for subsequent **GetNextPartition** requests. The *Flags* byte is a bit-mapped field used by Card Services to maintain state information for subsequent **GetNextPartition** requests. The *Link Offset* and *CIS Offset* fields are used by Card Services to maintain state information for subsequent **GetNextPartition** requests.

The *Card Offset* field is set by Card Services. It is the offset on the card where this partition begins. For CardBus PC Cards, this offset is relative to the indicated Base Address Register.

The *Part Size* is the total size of the partition.

The *EffBlockSize* field is the effective erase block size based on the device erase block size and how devices satisfy memory card accesses. If one device supplies the odd byte and another even bytes, the effective erase block size is twice the device erase block size.

The *JEDEC ID* field is the JEDEC Identifier of the devices in this region. If no *JEDEC ID* tuple is present in the CIS, this field is set to zero (0) by Card Services.

The lower fifteen bits of the *Part Type* field is a constant describing the type of partition. A zero (0) value means no partition information is available. A value of 7FFFH means the partition type is defined but unknown. Any other value is provided by the *TPLORG_TYPE* and *TPLORG_DESC* fields of the CISTPL_ORG tuple. (See the ***Metaformat Specification***.) The upper bit of the *Part Type* field indicates if the partition has EDC information. It will be set if the error detection code type in the *TPLFMT_EDC* field of the CISTPL_FORMAT tuple is non-zero. Clients may ignore the partition if they are not prepared to deal with EDCs. If a client can handle EDCs, it should use the Card Information Structure processing services to recover detailed EDC information. The client must first locate the appropriate format tuple and then process the EDC information.

The *PartMultiple* is the minimum size that may be used for a partition within the device space. It is expressed as a number of effective block sizes. For example, if the *EffBlockSize* field is 128 KBytes and the *PartMultiple* field is four (4), the actual minimum partition size is 512 KBytes. In addition, partition sizes greater than this minimum must be a multiple of this value. *PartMultiple* for most regions is related to device sizes rather than erase block sizes, since partitions may not cross devices which have interblock interactions within a device. As with *EffBlockSize*, the *PartMultiple* value accounts for the interleaving of multiple devices. The *PartMultiple* field is included for completeness. It saves the client the overhead of determining which region the partition lies in and obtaining this information from a **GetFirst/NextRegion** request.

WARNING:

Partitions which contain more than one type of device may require special Memory Technology Drivers (MTDs). Clients should use care in creating partitions which span multiple device types. Partitions that span multiple device types may not be usable in all systems.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirty-four (34)
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No partitions on PC Card

5.16 GetFirstRegion (06H)

CardServices(GetFirstRegion, *null/MTDHandle, null, ArgLength, ArgPointer*)

This service returns device information for the first region of devices on the card in the specified socket. Card Services obtains this information by directly accessing the PC Card's CIS. If there are no regions, the *Status* argument is set to NO_MORE_ITEMS. The *ClientHandle* for the MTD supporting access to this region is returned in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Region Attributes Field
4	TupleMask	1	O	N	Bit-mapped (defined below)
5	Access Speed	1	O	N	Window Speed Field
6	Flags	2	O	N	CS Region Flags Data
8	Link Offset	4	O	N	CS Region Link Data
12	CIS Offset	4	O	N	CS Region CIS Data
16	Card Offset	4	O	N	Card Memory Region Offset
20	Region Size	4	O	N	Region Size
24	EffBlockSize	4	O	N	Erase Block Size
28	PartMultiple	2	O	N	Partition Multiple (Erase Block units)
30	JEDEC ID	2	O	N	Region JEDEC Memory ID Code

The fields in this argument packet are the same as in **GetFirstPartition**. The *Flags*, *Link Offset* and *CIS Offset* fields in the argument packet described above are for internal use by Card Services. Card Services initialized them to the appropriate values. The client must preserve these values for subsequent **GetNextRegion** requests. The *Virtual Partition* attribute bit is interpreted as a virtual region attribute for regions.

Note: This service requires a PC Card be initialized with a CIS. This service does NOT interact with MTDs to determine region information.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirty-two (32)
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No regions or CIS on PC Card

5.17 GetFirstTuple (07H)

CardServices(GetFirstTuple, *null, null, ArgLength, ArgPointer*)

The **GetFirstTuple** service is no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service returns the first tuple of the specified type in the CIS for the specified socket. If there are no tuples, the *Status* argument is set to NO_MORE_ITEMS.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Desired Tuple	1	I	N	Desired Tuple Code Value
5	Reserved	1	I	ZERO	RESERVED (Reset to zero)
6	Flags	2	O	N	CS Tuple Flags data
8	Link Offset	4	O	N	CS Link State Information
12	CIS Offset	4	O	N	CS CIS State Information
16	Tuple Code	1	O	N	Tuple found
17	Tuple Link	1	O	N	Link value for tuple found

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bit 0	Return link tuples (set = true)
Bits 1 ·· 15	RESERVED (Reset to zero).

The following tuples are link tuples and will not be returned by this service unless Bit 0 of the *Attributes* field is set to one:

CISTPL_NULL,

CISTPL_LONGLINK_A,

CISTPL_LONGLINK_C,

CISTPL_LONGLINK_CB,

CISTPL_LONGLINK_MFC,

CISTPL_LINKTARGET,

CISTPL_NO_LINK, and

CISTPL_END

The *Desired Tuple* field is the tuple value desired. If it is 0FFH (255), the very first tuple of the CIS is returned (if it exists). If the *Desired Tuple* field is any other value on entry, the CIS is parsed attempting to locate a tuple which matches.

The *Flags*, *Link Offset* and *CIS Offset* fields in the above argument packet description are for internal use by Card Services. Card Services initializes them to the appropriate values. The client should preserve these values for subsequent **GetNextTuple** requests. The *Flags* field is used by Card Services to maintain state information during CIS processing requests. The *Link Offset* field and the *CIS Offset* field are also used by Card Services to maintain state information during CIS processing requests.

The *Tuple Code* and *Tuple Link* fields are the values returned from the tuple found.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eighteen (18)
BAD_SOCKET	*Socket* or function is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No Card Information Structure (CIS) or desired tuple not found

See also the ***Metaformat Specification***.

5.18 GetFirstWindow (37H)

CardServices(GetFirstWindow, *null/WindowHandle, null, ArgLength, ArgPointer*)

This service returns a window handle and associated window information for the first memory or I/O window of the specified socket and PC Card function. **GetNextWindow** is used to retrieve any additional windows for the specified socket. Page information for memory windows is obtained by **GetMemPage**. NO_MORE_ITEMS is returned if this socket does not have any allocated windows.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	O	N	Window Attribute Field
4	Base	4	O	N	System Base Address
8	Size	4	O	N	Window Size
12	AccessSpeed or IOAddrLines	1	O	N	Window Speed Field or number of I/O address lines decoded for 16-bit PC Card I/O windows

The *Socket*, *Attributes*, *Base*, *Size* and *AccessSpeed* fields are defined the same as in **RequestWindow**.

GetFirstWindow returns valid window *WindowHandles* for I/O range resources allocated via **RequestIO**.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirteen (13).
BAD_SOCKET	*Socket* is invalid.
NO_CARD	No PC Card in socket.
NO_MORE_ITEMS	No Windows open for this socket.
UNSUPPORTED_SERVICE	This service is not supported.

5.19 GetMemPage (39H) [16-bit PC Card only]

CardServices(GetMemPage, *WindowHandle, null, ArgLength, ArgPointer*)

This service returns the page information for the specified page of the requested memory window. The *WindowHandle* returned by **GetFirstWindow** or **GetNextWindow** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Card Offset	4	O	N	Card Offset Address
4	Page	1	I	N	Page Number

The *Card Offset* and *Page* fields are defined the same as in **MapMemPage**.

This service is only used for memory windows. If the *WindowHandle* identifies an I/O window, this service returns BAD_HANDLE.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to five (5).
BAD_HANDLE	*WindowHandle* is invalid or specified window is an I/O window.
UNSUPPORTED_SERVICE	This service is not supported.

5.20 GetNextClient (2AH)

CardServices(GetNextClient, *ClientHandle/ClientHandle, null, ArgLength, ArgPointer*)

This service returns the *ClientHandle* for the next registered client. The *ClientHandle* previously returned by **GetFirstClient** or **GetNextClient** is passed in the *Handle* argument. The next *ClientHandle* is returned in the *Handle* argument. If there are no more clients, the *Status* argument is set to NO_MORE_ITEMS.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped field (defined below)

When the *Attributes* Bit 0 is set (1), the *Socket* field is used to qualify the *ClientHandle*s considered. The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The bits are defined as follows:

Bit 0	0 = consider all registered clients 1 = consider clients for this socket only
Bits 1 ·· 15	RESERVED (Reset to zero)

If Bit 0 is set to one (1), only the clients accepting events for this socket and function are returned. If Bit 0 is reset to zero (0), all clients registered with Card Services are returned.

WARNING:

*If another client performs a successful **RegisterClient** or **DeregisterClient** request between a **GetFirstClient** and **GetNextClient** or two **GetNextClient** requests, the results are not predictable.*

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No more clients are registered

5.21 GetNextPartition (08H)

CardServices(GetNextPartition, *null, null, ArgLength, ArgPointer*)

This service returns device information for the next partition on the card in the specified socket based on the PC Card's CIS. If there are no more partitions, the *Status* argument is set to NO_MORE_ITEMS.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Partition Attributes Field
4	TupleMask	1	O	N	Bit-mapped field
5	Access Speed	1	O	N	Window Speed Field
6	Flags	2	I/O	N	CS Partition Flags Data
8	Link Offset	4	I/O	N	CS Partition Link Data
12	CIS Offset	4	I/O	N	CS Partition CIS Data
16	Card Offset	4	O	N	Card Memory Region Offset
20	Part Size	4	O	N	Partition Size
24	EffBlockSize	4	O	N	Erase Block Size
28	PartMultiple	2	O	N	Partition Multiple (Erase Block units)
30	JEDEC ID	2	O	N	Partition JEDEC Memory ID Code
32	PartType	2	O	N	Partition Type Field

The *Flags*, *Link Offset* and *CIS Offset* fields in the above argument packet description are for internal use by Card Services. Card Services initializes them during a **GetFirstPartition** or previous **GetNextPartition** request. If necessary, Card Services updates them during this request. The client must preserve these values between **GetNextPartition** requests. The *Socket* field must be the same as the original **GetFirstPartition** request. The *Flags* byte is a bit-mapped field used by Card Services to maintain state information for subsequent **GetNextPartition** requests. The *Link Offset* and *CIS Offset* fields are used by Card Services to maintain state information for subsequent **GetNextPartition** requests. The *Attributes* field must be the same as the original **GetFirstPartition** request.

SUCCESS is returned if there is another partition on the card. Other return codes are the same as **GetNextTuple**.

WARNING:

Partitions which contain more than one type of device may require special Memory Technology Drivers (MTDs). Clients should use care in creating partitions which span multiple device types. Partitions that span multiple device types may not be usable in all systems.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirty-four (34)
BAD_ARGS	Data from prior **GetFirst/Next** is corrupt
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No more partitions on PC Card

5.22 GetNextRegion (09H)

CardServices(GetNextRegion, *null/MTDHandle, null, ArgLength, ArgPointer*)

This service returns device information for the next region of devices on the card in the specified socket based on the PC Card's CIS. If there are no more regions, the *Status* argument is set to NO_MORE_ITEMS. The *ClientHandle* for the MTD supporting access to this region is returned in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Region Attributes Field
4	TupleMask	1	O	N	Bit-mapped field
5	Access Speed	1	O	N	Window Speed Field
6	Flags	2	I/O	N	CS Partition Flags Data
8	Link Offset	4	I/O	N	CS Partition Link Data
12	CIS Offset	4	I/O	N	CS Partition CIS Data
16	Card Offset	4	O	N	Card Memory Region Offset
20	Region Size	4	O	N	Region Size
24	EffBlockSize	4	O	N	Erase Block Size
28	PartMultiple	2	O	N	Partition Multiple (Erase Block units)
30	JEDEC ID	2	O	N	Partition JEDEC Memory ID Code

The *Attributes*, *Flags*, *Link Offset* and *CIS Offset* fields in the above argument packet description are for internal use by Card Services. Card Services initializes them during a **GetFirstRegion** or previous **GetNextRegion** request. If necessary, Card Services updates them during this request. The client must preserve these values between **GetNextRegion** requests. The *Socket* field must be the same as the original **GetFirstRegion** request. The *Flags* byte is a bit-mapped field used by Card Services to maintain state information for subsequent **GetNextRegion** requests. The *Link Offset* and *CIS Offset* fields are used by Card Services to maintain state information for subsequent **GetNextRegion** requests. The *Attributes* field must be the same as the original **GetFirstRegion** request.

SUCCESS is returned if there is another region on the card. Other return codes are the same as **GetNextTuple**.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirty-two (32)
BAD_ARGS	Data from prior **GetFirst/Next** is corrupt
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	Socket is invalid

5.23 GetNextTuple (0AH)

CardServices(GetNextTuple, *null, null, ArgLength, ArgPointer*)

The **GetNextTuple** service is no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service returns the next tuple of the specified type in the CIS for the specified socket. If there are no more tuples, the *Status* argument is set to NO_MORE_ITEMS.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped
4	Desired Tuple	1	I	N	Desired Tuple Code Value
5	Reserved	1	I	ZERO	Reserved (reset to zero)
6	Flags	2	I/O	N	CS Tuple Flags data
8	Link Offset	4	I/O	N	CS Link State Information
12	CIS Offset	4	I/O	N	CS CIS State Information
16	Tuple Code	1	O	N	Tuple found
17	Tuple Link	1	O	N	Link value for tuple found

The *Flags*, *Link Offset* and *CIS Offset* fields in the above argument packet description are for internal use by Card Services. They must be the same values returned by a **GetFirstTuple** or previous **GetNextTuple** request. They will be updated by Card Services. Their exit values should be preserved by the client for subsequent **GetNextTuple** requests. The *Socket* field describes the logical socket containing the desired card. The *Flags* field, *Link Offset* field, and ***CIS Offset*** field are used by Card Services to maintain state information during CIS processing requests.

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bit 0	Return link tuples (set = true)
Bits 1 ·· 15	RESERVED (Reset to zero).

The following tuples are link tuples and will not be returned by this service unless Bit 0 of the *Attributes* field is set to one:

CISTPL_NULL,

CISTPL_LONGLINK_A,

CISTPL_LONGLINK_C,

CISTPL_LONGLINK_CB,

CISTPL_LONGLINK_MFC,

CISTPL_LINKTARGET,

CISTPL_NO_LINK, and

CISTPL_END

The *Desired Tuple* field is the tuple value desired. If the field is set to 0FFH (255), the very next tuple of the CIS is returned (if it exists). If the *Desired Tuple* field is any other value, the CIS is parsed from the location returned by the previous **GetFirst/NextTuple** request attempting to locate a tuple which matches.

SUCCESS is returned if the specified tuple was found. If a specific tuple type was specified and it could not be located, NO_MORE_ITEMS is returned. If there was no CIS present, NO_MORE_ITEMS is returned. If no card is in the socket, NO_CARD is returned. If the entire CIS has been processed, NO_MORE_ITEMS is returned. Continuing to call **GetNextTuple** after receiving an NO_MORE_ITEMS return code results in more NO_MORE_ITEMS return codes.

The *Tuple Code* and *Tuple Link* fields are the values returned from the tuple found.

GetTupleData can be used to retrieve the actual tuple data.

Note: **GetNextTuple** attempts to match the contents of the *Desired Tuple* field. If the next physical tuple in the chain is desired, the client should ensure the *Desired Tuple* field is 0FFH (255) prior to invoking **GetNextTuple**. The *Desired Tuple* field may be modified between **GetFirst/NextTuple** requests.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eighteen (18)
BAD_ARGS	Data from prior **GetFirst/NextTuple** is corrupt
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	Desired tuple not found

See also **GetFirstTuple**, **GetTupleData**.

5.24 GetNextWindow (38H)

CardServices(GetNextWindow, *WindowHandle/WindowHandle, null, ArgLength, ArgPointer*)

This service returns the window information for the next window configured for this socket and PC Card function. NO_MORE_ITEMS is returned when no additional window information exists. The initial call to this service is with the *WindowHandle* and *Socket* returned by the **GetFirstWindow** service. Subsequent calls to this service must use the *WindowHandle* and *Socket* values returned by the previous call.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	O	N	Window Attribute Field
4	Base	4	O	N	System Base Address
8	Size	4	O	N	Window Size
12	AccessSpeed or IOAddrLines	1	O	N	Window Speed Field or number of I/O address lines decoded for 16-bit PC Card I/O windows

The *Socket*, *Attributes*, *Base*, *Size* and *AccessSpeed* fields are defined the same as in **RequestWindow**.

GetNextWindow returns valid window *WindowHandles* for I/O range resources allocated via **RequestIO**.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirteen (13).
BAD_HANDLE	*WindowHandle* is invalid.
BAD_SOCKET	*Socket* is invalid.
NO_CARD	No PC Card in socket.
NO_MORE_ITEMS	No additional Window information available.
UNSUPPORTED_SERVICE	This service is not supported.

5.25 GetStatus (0CH)

CardServices(GetStatus, *null, null, ArgLength, ArgPointer*)

This service returns the current status of a PC Card and its socket.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	CardState	2	O	N	Card State Output Data
4	SocketState	2	O	N	Socket State Output Data

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *CardState* field is the bit-mapped output data obtained from Socket Services. The bits identify the current state of the installed PC Card. They are:

Bit 0	Write Protected (set = true)
Bit 1	Card Locked
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Voltage Detect 1 (set = dead)
Bit 5	Battery Voltage Detect 2 (set = warning)
Bit 6	READY (set = ready asserted)
Bit 7	Card Detected (set = true)
Bit 8	Extended Status ReqAttn (set = true)
Bit 9	Extended Status RsvdEvt1 (set = true)
Bit 10	Extended Status RsvdEvt2 (set = true)
Bit 11	Extended Status RsvdEvt3 (set = true)
Bit 12 ·· 13	**Vcc** Level 0 = 5 volts **Vcc** indicated 1 = 3.3 volts **Vcc** indicated 2 = Reserved for X.X volts **Vcc** 3 = Reserved, not used.
Bits 14 ·· 15	RESERVED (Reset to zero)

This information is obtained from the Socket Services **GetStatus** and **GetSocket** services. The *CardState* field is created from the Socket Services *CardState*, *Vcontrol*, and *State* fields. If an I/O card is installed in the specified socket, card state is returned from the Pin Replacement Register and the Extended Status register (if present). If certain state bits are not present in the pin replacement or Extended Status registers (see the ***Electrical Specification***), a simulated state bit value is returned as defined below:

WP (Write Protected)	Not write protected
BVD1 (Battery Voltage Detect 1)	Power Good
BVD2 (Battery Voltage Detect 2)	Power Good
READY	Ready
ReqAttn	Reset to 0
RsvdEvt1	Reset to 0
RsvdEvt2	Reset to 0
RsvdEvt3	Reset to 0

The *SocketState* field is the bit-mapped output data returned from Socket Services. These bits identify the current socket state. They are:

Bit 0	Write Protect Change
Bit 1	Card Lock Change (set = true)
Bit 2	Ejection Request Pending (set = true)
Bit 3	Insertion Request Pending (set = true)
Bit 4	Battery Dead Change (set = true)
Bit 5	Battery Warning Change (set = true)
Bit 6	Ready Change (set = true)
Bit 7	Card Detect Change (set = true)
Bits 8 ·· 15	RESERVED (Reset to zero)

This information is obtained from the Socket Services **GetSocket** service for memory cards. The *SocketState* field is created from the Socket Services Socket Attributes.

SUCCESS is returned if the *Socket* field is valid.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to six (6)
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)

5.26 GetTupleData (0DH)

CardServices(GetTupleData, *null, null, ArgLength, ArgPointer*)

The **GetTupleData** service is no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service returns the content of the last tuple returned by **GetFirst/NextTuple**. The tuple data returned is packed so that all tuple data bytes are contiguous and not even byte only (even if the data came from attribute memory address space). This service only returns data bytes from the tuple body. The tuple code and link fields are never returned in the *Tuple Data* field.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Desired Tuple	1	I	N	Desired Tuple Code Value
5	Tuple Offset	1	I	N	Offset into tuple from link byte
6	Flags	2	I/O	N	CS Tuple Flags data
8	Link Offset	4	I/O	N	CS Link State Information
12	CIS Offset	4	I/O	N	CS CIS State Information
16	Tuple Data Max	2	I	N	Maximum size of tuple data area
18	Tuple Data Len	2	O	N	Number of bytes in tuple body
20	Tuple Data	N	O	N	Tuple Data

The argument packet has been structured to use the same fields as **GetFirst/NextTuple**. This allows a client to locate a tuple with those services and then retrieve tuple data with this service using the same argument packet.

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes, Desired Tuple, Flags, Link Offset,* and *CIS Offset* fields are for internal use by Card Services. They must be the same values returned by a **GetFirstTuple** or previous **GetNextTuple** request. Their exit values should be preserved by the client for subsequent **GetNextTuple** requests.

The *Socket* field describes the logical socket containing the desired card. The *Flags* byte**,** *Link Offset* field, and *CIS Offset* field are used by Card Services to maintain state information during CIS processing requests. The *Attributes* and *Desired Tuple* fields describe the tuple being processed.

The *Tuple Offset* field allows partial tuple information to be retrieved starting anywhere within the tuple. The actual number of tuple bytes in the tuple body is returned in the *Tuple Data Len* field. This value will be larger than *Tuple Data Max* if the tuple body is larger than the space provided in *Tuple Data*. Attempting to read beyond the end of a tuple returns with a return code of NO_MORE_ITEMS.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is less than twenty (20)
BAD_ARGS	Data from prior **GetFirst/NextTuple** is corrupt
BAD_SOCKET	*Socket* or function is invalid
NO_CARD	No PC Card in socket
NO_MORE_ITEMS	No more tuple data on PC Card

5.27 InquireConfiguration (3DH)

CardServices(**InquireConfiguration**, *null, null, ArgLength, ArgPointer*)

This service returns one of a function's possible configurations as defined in the function's CIS by the **CISTPL_CONFIG** and **CISTPL_CFTABLE_ENTRY** tuples. The **InquireConfiguration** structure pointed to by the *ArgPointer* argument is composed of a fixed length header followed by three variable length sections. The header contains three fields which identify where each of the variable length sections begin in the data returned in the buffer pointed to by the *ArgPointer* argument.

Header (multiple fields described below)
Offset of Configuration Register Structure
Offset of Array of Tagged Resource Structures for Default Configuration
Offset of Array of Tagged Resource Structures for Selected Configuration
Request Type
Configuration Register Structure
Array of Tagged Resource Structures for Default Configuration
Array of Tagged Resource Structures for Selected Configuration

Following is a detailed view of the *Header* portion of the **InquireConfiguration** service data structure:

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2 ·· 17	Reserved	16	I/O	N	Reserved (Initialized by Card Services during FIRST requests, must be preserved for subsequent CURRENT and NEXT requests. Card Services uses this space when making **GetFirstTuple** and **GetNextTuple** service requests as needed to support **InquireConfiguration** service requests.)
18	Actual Size	2	O	N	Size of the **InquireConfiguration** structure. If this value is greater than the *ArgLength* argument, then the buffer provided is too small to hold the **InquireConfiguration** structure.
20	Config Offset	2	I/O	N	Offset of Configuration Registers Structure in returned data (Initialized by Card Services during FIRST request processing, must be preserved for subsequent CURRENT and NEXT requests)
22	Default Offset	2	I/O	N	Offset of array of tagged resource structures for default configuration in returned data (Initialized by Card Services during FIRST request processing, must be preserved for subsequent CURRENT and NEXT requests)
24	Selected Offset	2	I/O	N	Offset of array of current tagged resource structures for selected configuration in returned data (Set by Card Services for all requests)
26	Request Type	1	I	N	Type of request: FIRST, CURRENT or NEXT

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0)

value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Request Type* field determines which configuration description the client is requesting. This field must be set to FIRST for the initial request by a client to read a function's possible configurations. To refresh the selected configuration description requested by the last **InquireConfiguration** request, the client sets this field to REFRESH. To retrieve the description of the next configuration in sequence, the client sets the *Request Type* field to NEXT. To retrieve the current configuration of a PC Card function and socket, the client sets the *Request Type* field to CURRENT. The *Request Type* field is defined as follows :

Bits 0 ·· 1	Requested Configuration: 0 - FIRST 1 - REFRESH 2 - NEXT 3 - CURRENT
Bits 2 ·· 7	Reserved (must be reset to zero)

Many of the fields in the header area must be preserved by the client between Card Services requests. The individual fields are identified in the columns describing those fields.

The Configuration Registers structure describes the function's configuration registers. Much of this information is common to all of the function's configurations.

Offset	Field	Size	Type	Value	Detail/Description
n	ConfigBase	4	O	N	Base address for function's configuration registers
n + 4	Features	1	O	N	The lower nibble is the number of bytes of configuration register presence bytes. This value ranges from 1 to 16 to represent one to 128 configuration registers. The next two significant bits represent the number of custom interfaces supported by this function. This value ranges from zero to four. If this value is non-zero, the following fields are an array of four-byte custom interface identifiers, one per custom interface. The next bit is reserved and is reset to zero. The most significant bit indicates whether this is a 16-bit or CardBus PC Card. If reset, it is a 16-bit PC Card. If set, the card is a CardBus PC Card.
n + 5	Presence Bytes	1 ·· 16	O	N	Each byte is bit-mapped representing the presence of defined configuration registers. The bytes are stored in little endian order with the first byte representing the first eight possible configuration registers. If a bit is set the register is present. If the bit is reset, the register is not present.
*	Array of Custom Interfaces	0 ·· 16	O	N	Zero to four, four-byte custom interface identifiers representing the custom interfaces used by the function's configurations. This field is ignored for CardBus PC Cards.

The last two sections of the buffer pointed to by the *ArgPointer* argument contain an array of tagged resource structures describing the function's default and selected configurations. The tagged resource structures all use a consistent format with the first byte identifying the resource type and the second

byte the length of the structure. The information in these structures is derived from the descriptor fields of the function's Configuration tuple.

Offset	Field	Size	Type	Value	Detail/Description
0	Tag Field	1	I	N	Tag Field Byte
1	Length	1	I	N	Length of current Resource Structure

Within the selected configuration description, each *Tag Field* is bit-mapped as follows:

Bit 0 ·· 7	Tag Field Number (7 bits = 127 possible tags)
Bit 8	Flag Bit : 0 = Explicit : Resource read directly from the current Configuration tuple in the CIS 1 = Default : A default resource value as specified by CIS

In order to conserve space in the function's Card Information Structure (CIS), resource requirements from a previous configuration are often used in a current configuration. The default configuration tracks the current default configuration resource requirements. If these requirements are not overridden by the selected configuration, they are repeated in the array of tagged resource structures in the current configuration section of the returned data.

All resource requirements for a returned configuration are completely described by the section containing the array of tagged resource structures for the selected configuration. The array of tagged resource structures for the default configuration is only maintained for Card Services use. Client device drivers are not expected to utilize this information.

Client device drivers can determine if a particular tagged resource structure is included in the selected configuration by default according to the upper bit of the first byte of the tagged resource structure. If the structure is being included by default, this bit is set. If the resource is described by the current configuration tuple, this bit is reset.

The returned data also indicates whether the described resource was available when the structure was created. How this information is returned depends on the type of tagged resource structure. An array of tagged resource structures is terminated by a resource type of 7FH. The following resource types are defined:

TAG (HEX)	Resource Type
00	16-bit Interface
01	I/O
02	Window
03	IRQ
04	Reserved[1]
05	Timing
06	Power
07	Thermal
08	CardBus Interface
09 ·· 7E	Reserved (do not use)
7F	End of array of tagged resources structures

1. Legacy feature no longer supported.

16-bit Interface Tagged Resource Structure

The 16-bit Interface tagged resource structure identifies the values written to the function's Configuration Option Registers of a 16-bit PC Card to select the configuration, the interface and the miscellaneous features required to use the function.

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	00H or 80H	Type = 16-bit Interface Resource
1	Length	1	N	N	Total length of structure
2	Option Value	1	N	N	Bit-mapped (defined below)
3	Interface	1	N	N	For 16-bit PC Cards, see the ***TPCE_IF: Interface Description Field*** section of the ***Metaformat Specification***.
4	Miscellaneous Features	1	N	N	Bit-Mapped (defined below)

The *Option Value* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	Option Not Available in Host	Value to be written to Configuration Option Register of a 16-bit PC Card to select this configuration.						

The *Miscellaneous Features* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	Audio Setting	MaxTwins Setting			Audio	MaxTwins		

MaxTwins is the value taken from the TPCE_MI MaxTwins field of the function's CIS. For a complete description of this field please refer to the ***Metaformat Specification***. This value must be preserved for a **ConfigureFunction** request.

Audio is the value taken from the TPCE_MI Audio field of the function's CIS. For a complete description of this field please refer to the ***Metaformat Specification***. This value must be preserved for a **ConfigureFunction** request.

The *MaxTwins Setting* field is the value of the MaxTwins setting for the function's COR set.

The *Audio Setting* field is the value of the Audio setting for the function's COR set.

I/O Tagged Resource Structure

The I/O Tagged Resource Structure is used to describe a configuration's I/O resource requirements. The structure is defined as follows:

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	01 or 81H	Type = I/O Resource
1	Length	1	N	N	Total length of structure
2	Num Ranges	1	N	N	1 .. 16 I/O Address Ranges
3	I/O Ranges	*	*	*	Array of I/O Ranges

Each I/O Range within the I/O Tagged Resource structure has the following format. The fields within each I/O Range use the same encoding as the fields of the **RequestWindow** *ArgPacket* described in Section 5.53 of the ***Card Services Specification.***

n	Attributes	2	I/O	N	Range Attribute field
n + 2	Base	4	I/O	N	Base address of range in host system address space
n + 6	Size	4	I/O	N	Size of Range
n + 10	IOAddrLines	1	I	N	Number of I/O address lines decoded for 16-bit PC Card I/O windows
n + 11	Flags	1	O	N	Bit-mapped (defined below)

The *Flags* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	I/O Range Not Available

Window Tagged Resource Structure

The Window Tagged Resource Structure is used to describe a configuration's memory and/or I/O window resource requirements. The structure is defined as follows:

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	02 or 82H	Type = Window Resource
1	Length	1	N	N	Total length of structure
2	Num Ranges	1	N	N	1 .. 8 Window Address Ranges
3	Memory Ranges	*	*	*	Array of Window Ranges

Each Window Range within the Window Tagged Resource structure has the following format. The fields *Attributes* through *Speed* within each Window Range use the same encoding as the fields of the **RequestWindow** *ArgPacket* described in Section ***5.53 RequestWindow (21h)***.

n	Attributes	2	I/O	N	Range Attribute field
n + 2	Base	4	I/O	N	Base address of range in host system address space
n + 6	Size	4	I/O	N	Size of Range
n + 10	Speed	1	I	N	Access speed for range
n + 11	Flags	1	O	N	Bit-mapped (defined below)
n + 12	CardOffset	4	O	N	Card Offset

The *Flags* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	RFU (0)	Window Range Not Available

The *CardOffset* field describes the address of the card-offset defined for the configuration.

IRQ Tagged Resource Structure

The IRQ Tagged Resource structure is used to describe a configuration's interrupt routing recommendation. The fields within the structure use the same encoding as the fields of the **RequestIRQ** *ArgPacket* described in Section 5.51 of the ***Card Services Specification.***

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	03 or 83H	Type = IRQ Resource
1	Length	1	N	N	Total length of structure
2	Attributes	2	I/O	N	Bit-mapped (see definition in **RequestIRQ** service). When bit 15 is SET (1), the IRQ or set of IRQs indicated is not available.
4	AssignedIRQ	1	O	N	IRQ Number Assigned by CS when used with the **ConfigureFunction** service. Ignore for **InquireConfiguration**.
5	IRQInfo1	1	I	N	First TPCE_IR IRQ Byte
6	IRQInfo2	2	I	N	Optional TPCE_IR IRQ bytes

Timing Tagged Resource Structure

The Timing Tagged Resource structure is used to describe a configuration's Timing requirements.

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	05 or 85H	Type = Timing Resource
1	Length	1	N	N	Total length of structure
2	Max WAIT	2	N	N	Maximum WAIT time in ms
4	Max Busy	2	N	N	Maximum BUSY time in ms

Power Tagged Resource Structure

The Power Tagged Resource structure is used to describe a configuration's Power requirements.

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	06 or 86H	Type = Power Resource
1	Length	1	N	N	Total length of structure
2	**V**cc	1	I	N	Vcc Setting
3	**V**pp**1**	1	I	N	Vpp1 Setting
4	**V**pp**2**	1	I	N	Vpp2 Setting
5	Attributes	1	I	N	Bit-mapped (defined below)
6	I_{TOT} Static	2	I	N	Total Static Current
8	I_{TOT} Avg	2	I	N	Total Average Current
10	I_{TOT} Peak	2	I	N	Total Peak Current
12	I_{TOT} Pdwn	2	I	N	Total Power Down Current
14	Flags	1	I	N	Bit-mapped (defined below)

The *Vcc*, *Vpp1*, and *Vpp2* fields all represent voltages expressed in tenths of volts. Since these fields are a byte wide, values from 0.0 to 25.5 Volts may be reported.

The *Attributes* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	RFU (0)	Assumed Pdwn I	Assumed Peak I	Assumed Avg I	Assumed Static I	RFU (0)	RFU (0)	RFU (0)

If an *Assumed* bit is SET (1), then the corresponding Current (I_{TOT}) value was not specified explicitly in the function's CIS; therefore, the value is assumed as outlined in the following tables:

PC Cards with a Card Information Structure compliant with the ***PC Card Standard February 1995 Release*** or later (see CISTPL_VERS_1 in the ***Metaformat Specification***) use the following assumptions:

Voltage	Static (mA)	Average (mA)	Peak (mA)
3.3V Vcc	70	70	70
5.0V Vcc	100	100	100
3.3V Vpp	60	60	60
5.0V Vpp	60	60	60
12.0V Vpp	60	60	60

PC Cards with a Card Information Structure compliant to versions of the Standard prior to the ***PC Card Standard***, February 1995, (see CISTPL_VERS_1 in the ***Metaformat Specification***) use the following assumptions:

Voltage	Static (mA)	Average (mA)	Peak (mA)
3.3V Vcc	300	500	750
5.0V Vcc	300	500	750
3.3V Vpp	60	60	60
5.0V Vpp	60	60	60
12.0V Vpp	60	60	60

The I_{TOT} fields all represent currents expressed in one hundred microamp (100 µA) increments. Since these fields are 2 bytes wide, values from 0.0 to 6.5535 Amps may be reported.

The *Flags* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	Power Down Setting	Pdwn I Not Available	Peak I Not Available	Avg I Not Available	Static I Not Available	Vpp2 Not Available	Vpp1 Not Available	Vcc Not Available

The *Power Down Setting* field reflects the Power Down bit in the TPCE_MI field of the funtion's CIS. This field is only applicable for 16-bit PC Cards. For CardBus PC Cards this field is ignored.

Thermal Tagged Resource Structure

The Thermal Tagged Resource structure is used to describe a configuration's thermal requirements.

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	07 or 87H	Type = Thermal Resource
1	Length	1	N	N	Total length of structure
2	Thermal Static	2	I	N	Static Power dissipation
4	Thermal Avg	2	I	N	Average Power Dissipation
6	Thermal Peak	2	I	N	Peak Power Dissipation
8	Thermal Pdwn	2	I	N	Power Down Power Dissipation
9	Attributes	1	O	N	Bit-mapped (defined below)
10	Flags	1	O	N	Bit-mapped (defined below)

The *Thermal* fields all represent power dissipation expressed in one milliwatt (1 mW) increments. Since these fields are 2 bytes wide, values from 0 to 65.535 Watts are possible.

The *Attributes* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	RFU (0)	Assumed Pdwn Thermal	Assumed Peak Thermal	Assumed Avg Thermal	Assumed Static Thermal	RFU (0)	RFU (0)	RFU (0)

If an *Assumed* bit is SET (1), then the corresponding *Thermal* value was not specified explicitly in the function's CIS; therefore, the value is calculated as the product of (Current(I) x Voltage(V)).

The *Flags* field is defined as follows:

Bit	7	6	5	4	3	2	1	0
Value	RFU (0)	Pdwn Thermal Not Available	Peak Thermal Not Available	Avg Thermal Not Available	Static Thermal Not Available	RFU (0)	RFU (0)	RFU (0)

CardBus Interface Tagged Resource Structure

The CardBus Interface Tagged Resource Structure identifies the values written to the function's Command Register to select the features required to use the function.

Offset	Field	Size	Type	Value	Detail/Description
0	Resource Type	1	N	08h or 88h	Type = CardBus Interface Resource
1	Length	1	N	N	Total length of structure
2	Misc Features	2	N	N	Bit-mapped (defined below)
4	Misc Features Settings	2	N	N	Bit-mapped (defined below)

The *Misc Features* field is the value of the TPCE_CBMI field of the function's CIS. For a complete description of this field please refer to the ***Metaformat Specification***. This value must be preserved for a **ConfigureFunction** request.

The *Misc Features Settings* field is the values of the function's Miscellaneous Features settings in the Command Register.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is invalid
BAD_SOCKET	*Socket* or function is invalid
CONFIGURATION_LOCKED	PC Card function already configured and *Request Type* not CURRENT
NO_CARD	No PC Card in socket
UNSUPPORTED_SERVICE	This service is not supported

5.28 MapLogSocket (12H)

CardServices(MapLogSocket, *null, null, ArgLength, ArgPointer*)

This service maps a Card Services logical socket to its Socket Services physical adapter and socket values.

Offset	Field	Size	Type	Value	Detail/Description
0	Log Socket	2	I	N	Logical Socket
2	Phy Adapter	1	O	N	Physical Adapter Number
3	Phy Socket	1	O	N	Physical Socket Number

The *Log Socket* field contains the socket to convert to Socket Services adapter and physical socket values.

The *Phy Adapter* field is returned by Card Services, if the *Log Socket* field is valid. It is the Socket Services adapter value to address the logical socket. Physical adapters and physical sockets adapters are numbered starting at zero (0) . The *Phy Socket* field is returned by Card Services, if the *Log Socket* field is valid. It is the Socket Services socket value to address the logical socket.

Note: This service is not expected to be required for most clients of Card Services. It is intended to provide additional information to system utilities which may require the ability to map between logical and physical resources.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_SOCKET	*Socket* is invalid

5.29 MapLogWindow (13H) [16-bit PC Card only]

CardServices(MapLogWindow, *WindowHandle, null, ArgLength, ArgPointer*)

This service maps a Card Services *WindowHandle* passed in the *Handle* argument to its Socket Services physical adapter and window.

Offset	Field	Size	Type	Value	Detail/Description
0	Phy Adapter	1	O	N	Physical Adapter Number.
1	Phy Window	1	O	N	Physical Window Number. This is the number of the physical window used to map a 16-bit PC Card.

The *Phy Adapter* field is the Socket Services physical adapter number containing the window. The *Phy Window* field is the Socket Services physical window number.

Note: This service is not expected to be required for most clients of Card Services. It is intended to provide additional information to system utilities which may require the ability to map between logical and physical resources.

WARNING:

Card Services may use more than one physical window as a logical window. Values returned by this service may only describe a part of the logical window.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
BAD_HANDLE	*WindowHandle* is not a valid 16-bit PC Card memory window

5.30 MapMemPage (14H) [16-bit PC Card only]

CardServices(MapMemPage, *WindowHandle, null, ArgLength, ArgPointer*)

This service selects the memory area on a PC Card into a page of a window allocated with the **RequestWindow** service. The *WindowHandle* returned by **RequestWindow** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Card Offset	4	I	N	Card Offset Address
4	Page	1	I	N	Page Number

The *Card Offset* field is the absolute offset from the beginning of the PC Card to map into system memory.

The *Page* field is the page number for the window. This page of the window in system memory address space will be mapped to the requested area of the PC Card. If the Paged bit in the *Attributes* field for a window is not set, this value must be zero (indicating the first and only page).

Service not valid for CardBus PC Cards.

This service is only used for memory windows. If the *WindowHandle* identifies an I/O window, this service returns BAD_HANDLE.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to five (5)
BAD_HANDLE	*WindowHandle* is invalid or the specified window is an I/O window
BAD_OFFSET	Offset is invalid
BAD_PAGE	Page is invalid
NO_CARD	No card in socket

5.31 MapPhySocket (15H)

CardServices(MapPhySocket, *null, null, ArgLength, ArgPointer*)

This service maps Socket Services physical adapter and socket values to a Card Services logical socket.

Offset	Field	Size	Type	Value	Detail/Description
0	Log Socket	2	O	N	Logical Socket
2	Phy Adapter	1	I	N	Physical Adapter Number
3	Phy Socket	1	I	N	Physical Socket Number

The *Log Socket* field is returned by Card Services. It contains the logical socket representing the *Phy Adapter* and *Phy Socket* provided. Physical adapters and physical sockets are numbered starting at zero (0). The *Phy Adapter* field and *Phy Socket* fields, if valid, are converted to a *Log Socket* value.

SUCCESS is returned if the *Phy Adapter* and *Phy Socket* fields are valid.

Note: This service is not expected to be required for most clients of Card Services. It is intended to provide additional information to system utilities which may require the ability to map between logical and physical resources.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_ADAPTER	Phy Adapter is invalid
BAD_SOCKET	*Phy Socket* is invalid

5.32 MapPhyWindow (16H) [16-bit PC Card only]

CardServices(MapPhyWindow, *null/WindowHandle, null, ArgLength, ArgPointer*)

This service maps Socket Services physical adapter and window values to a Card Services logical *WindowHandle.*

Offset	Field	Size	Type	Value	Detail/Description
0	Phy Adapter	1	I	N	Physical Adapter Number
1	Phy Window	1	I	N	Physical Window Number

SUCCESS is returned if the *Phy Adapter* and *Phy Window* fields are valid. BAD_ADAPTER or BAD_WINDOW is returned if the corresponding value is invalid.

Note: This service is not expected to be required for most clients of Card Services. It is intended to provide additional information to system utilities which may require the ability to map between logical and physical resources.

WARNING:

Card Services may use more than one physical window as a logical window. Values returned by this service may only describe a part of the logical window.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
BAD_ADAPTER	Adapter is invalid
BAD_WINDOW	Window is invalid

5.33 ModifyConfiguration (27H)

CardServices(ModifyConfiguration, *ClientHandle, null, ArgLength, ArgPointer*)

This service allows a socket and PC Card configuration to be modified without a pair of **Release/RequestConfiguration** services. The *ClientHandle* originally passed to **RequestConfiguration** is passed in the *Handle* argument. This service can only modify a configuration requested via **RequestConfiguration**.

I/O addresses mapped (either with **RequestIO** or **RequestWindow**) and IRQ routing can only be changed by first using **ReleaseConfiguration** and then using **Release/RequestIO/IRQ/Window** followed by **RequestConfiguration**.

VCC can not be changed using the **ModifyConfiguration** service. **VCC** may be changed by first invoking **ReleaseConfiguration** followed by **RequestConfiguration** with a new **VCC** value.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Reserved	1	I	N	Reserved
5	**VPP1**	1	I	N	**VPP1** Setting
6	**VPP2**	1	I	N	**VPP2** Setting

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bit 0	RESERVED (Reset to zero)
Bit 1	Enable IRQ Steering (set = true)
Bit 2	IRQ change valid (set = true)
Bit 3	Reserved—value ignored
Bit 4	**VPP1** change valid (set = true)
Bit 5	**VPP2** change valid (set = true)
Bit 6	RESERVED[1] (Reset to zero)
Bit 7	RESERVED[1] (Reset to zero)
Bit 8	RESERVED (Reset to zero)
Bit 9	VSOVERRIDE (set = override VS pins)
Bits 10 ·· 15	RESERVED (Reset to zero)

1. Legacy feature no longer supported

Enable IRQ Steering is set to one to connect the PC Card **IREQ#** to a previously selected system interrupt. *IRQ change valid* is set to one to request the IRQ steering enable to be changed. The **VPP1** *change valid* and **VPP2** *change valid* bits are set to one to request a change to the corresponding voltage level for the PC Card.

The **VPP1** and **VPP2** fields both represent voltages expressed in tenths of a volt. Since these fields are a byte wide, values from zero (0) to 25.5 volts may be set. To be valid, the exact voltage must be available through the system's Socket Services.

> *WARNING:*
>
> *Using this service to set **VPP1** or **VPP2** to zero (0) volts may result in the loss of a PC Card's state. Any client setting **VPP1** or **VPP2** to zero (0) volts is responsible for ensuring the PC Card's state is restored when power is re-applied to the card.*

After card insertion and prior to the first **RequestConfiguration** call for this client the voltage levels applied to the card will be those specified by the Card Interface Specification. For Low Voltage keyed cards, if a client desires to apply a voltage inappropriate for this card to any pin then the *VSOVERRIDE* bit must be set in the *Attribute* field otherwise a BAD_VCC or BAD_VPP will be returned.

> *WARNING:*
>
> *The VSOVERRIDE bit is provided for clients that have a need to override the information provided in the CIS. The Client must exercise caution when setting this bit as it overrides any voltage level protection provided by Card Services.*

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to seven (7)
BAD_ATTRIBUTE	IRQ steering cannot be disabled or enabled
BAD_HANDLE	*ClientHandle* doesn't match owning client
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
BAD_VPP	Requested **VPP1** or **VPP2** voltage is not available on socket
NO_CARD	No PC Card in socket

5.34 ModifyWindow (17H)

CardServices(ModifyWindow, *WindowHandle, null, ArgLength, ArgPointer*)

This service modifies the attributes, or access speed of a window previously allocated with the **RequestWindow** service. The *WindowHandle* returned by **RequestWindow** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Window Attributes Field
2	AccessSpeed	1	I	N	Window Speed

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to zero)
Bit 1	Memory type (set = attribute) - Must be reset (0) for CardBus PC Card
Bit 2	Enable (set = true, reset = disable)
Bit 3	AccessSpeed valid (set = true) - Must be reset (0) for CardBus PC Card
Bits 4 ·· 15	RESERVED (Reset to zero)

Attribute memory is not a valid characteristic of a CardBus PC Card, and thus, bit 1 of the *Attributes* field must never be set for these cards.

Note: *AccessSpeed valid* is set to one by the client when the *Access Speed* field has a value that the client wants set for the window. If *AccessSpeed valid* is reset to zero, the *Access Speed* field is ignored and the access speed for the window is not modified.

The *Access Speed* field is bit-mapped as follows:

Bits 0 ·· 2	Device speed code, if speed mantissa is zero Speed exponent, if speed mantissa is not zero
Bits 3 ·· 6	Speed mantissa
Bit 7	Wait (set = use **WAIT#**, if available)

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero, the lower bits are a binary code representing a speed from the following table:

Code	Speed
0	(Reserved - do not use)
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5 ·· 7	(Reserved - do not use)

The *WindowHandle* identifies the window for this request. It must be the value returned by the original **RequestWindow** request.

This service is only used for memory windows. If the *WindowHandle* identifies an I/O window, this service returns BAD_HANDLE.

Note: Only some of the 16-bit PC Card window attributes or the access speed field may be modified by this request. The **MapMemPage** service is also used to set the offset into a 16-bit PC Card memory to be mapped into system memory for paged windows. The Request/ReleaseWindow service must be used to change the window base or size.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to three (3)
NO_CARD	No PC Card in socket
BAD_ATTRIBUTE	Attributes are invalid or window cannot enabled/disabled
BAD_HANDLE	*WindowHandle* is invalid or the specified window is an I/O window
BAD_SPEED	Speed is invalid

5.35 OpenMemory (18H)

CardServices(OpenMemory, *ClientHandle/MemoryHandle, null, ArgLength, ArgPointer*)

This service opens an area of a memory card to allow use of the **Read/Write/CopyMemory** and the erase services. It associates an MTD and an absolute card offset with a *MemoryHandle*. Card Services will apply power to the socket if the socket was not being used. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument. The *MemoryHandle* returned in the *Handle* argument must be used in the **Read/Write/CopyMemory** and **EraseQueue** requests.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Attributes of memory area to be accessed
4	Offset	4	I	N	Card Offset for Area to Open

For 16-bit PC Cards, the *Socket* field describes the logical socket containing the desired card.

For CardBus PC Cards, the *Socket* field identifies the logical socket and function. The least significant byte is the logical socket. The most significant byte of the *Socket* field is the function. Allowable functions are numbered from 0 to 7.

The *Attributes* field is bit-mapped. It indicates the type of memory that is being opened as follows:

Bit 0	Memory type (set = attribute)
Bit 1	Exclusive (set = true)
Bit 2	RESERVED (Reset to zero)
Bit 3 ··4	Prefetchable / Cacheable Memory (set to one = true) 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use
Bits 5 ·· 12	RESERVED (Reset to zero)
Bits 13 ·· 15	Base Address Register number (1-7).

Bit 0 specifies whether attribute or common memory will be accessed. If attribute memory is being accessed, the client must explicitly access the memory correctly - this means that only even bytes can be reliably read and written. Bit 1 allows a client to gain exclusive access to the memory area beginning at the specified offset. Other clients that attempt to open the memory area beginning at the same offset will receive an IN_USE return code.

Attribute memory is not a valid characteristic of a CardBus PC Card, and thus, bit 0 of the *Attributes* field must never be set for these cards.

Prefetchable / Cacheable applies to CardBus PC Cards only. 16-bit PC Cards shall use zero (0) for this field.

The *Base Address Register* number indicates the associated Base Address Register on the CardBus PC Card. Base Address Register number seven (7) always refers to the Expansion ROM Base Address Register.

The *Offset* identifies the byte offset to the beginning of the portion of the card that the client will be accessing. Card Services uses this information to determine the correct *MemoryHandle* to return. The

offset is also saved by Card Services to adjust relative offsets supplied by the **Read/Write/CopyMemory** and the erase services to absolute offsets for MTDs.

The *MemoryHandle* field is returned by this request. It must be used for all subsequent read, write, copy and erase requests to the identified memory area. When all accesses have been performed, the client must perform a **CloseMemory** request with this *Handle*. Each **OpenMemory** increments a use count maintained for each region and each **CloseMemory** decrements this counter.

Only one **OpenMemory** needs to be performed for all accesses to a specific area of a PC Card by a single client.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eight (8)
BAD_HANDLE	Invalid *ClientHandle*
BAD_OFFSET	Offset is invalid
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket
IN_USE	Memory area is in-use, exclusively

5.36 ReadMemory (19H)

CardServices(ReadMemory, *MemoryHandle, buffer, ArgLength, ArgPointer*)

This service reads data from a PC Card via the specified *MemoryHandle.* An MTD is used to perform the actual read. The *MemoryHandle* returned by **OpenMemory** is passed in the *Handle* argument. The pointer to the system memory buffer that will receive the data read from the PC Card is passed in the Pointer argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Card Offset	4	I	N	Card Source Offset
4	Count	4	I	N	Number of bytes to transfer

The *Card Offset* is a relative offset from the physical offset provided to the **OpenMemory** request used to obtain the *MemoryHandle.* It is the first location on the PC Card where the data should be read.

The *Count* field is the number of bytes to read from the PC Card.

If the *MemoryHandle* identifies common memory, all bytes requested are transferred. If the *MemoryHandle* identifies attribute memory, the client must recognize that memory locations with odd addresses may not have valid data. The client is expected to access the attribute memory in an appropriate way.

When used in a processor mode that supports segmentation (e.g. x86 architecture systems in 286 protected mode operation), all bytes transferred are required to be contained within the segment referenced by the system memory buffer pointer. This limits the count requested to be less than or equal to the maximum segment size.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eight (8)
BAD_HANDLE	Invalid *MemoryHandle*
BAD_OFFSET	Invalid source offset
READ_FAILURE	Unable to read media
BAD_SIZE	Size of area to read is invalid
NO_CARD	No PC Card in socket

5.37 RegisterClient (10H)

CardServices(RegisterClient, *null/ClientHandle, ClientEntry, ArgLength, ArgPointer*)

This service registers a client with Card Services. The *ClientHandle* returned in the *Handle* argument must be passed to **DeregisterClient** when the client terminates. The Client callback handler entry point is passed in the Pointer argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined below)
2	Event Mask	2	I	N	Events to notify Client
4	Client Data	8	I	N	Data for the Client (binding specific)
12	Version	2	I	BCD	the *CSLevel* this client expects

The *Attributes* field is bit-mapped. It identifies the type of client registering and what type of artificial **CARD_INSERTION** notifications Card Services should generate. The field is defined as follows:

Bit 0	Memory client device driver (set = true)
Bit 1	Memory Technology Driver (set = true)
Bit 2	I/O client device driver (set = true)
Bit 3	**CARD_INSERTION** events for sharable PC Cards (set = true)
Bit 4	**CARD_INSERTION** events for cards being exclusively used (set = true)
Bits 5 ·· 15	RESERVED (reset to zero)

Bits 0, 1 and 2 are mutually exclusive, only one bit may be set to one, but one of the bits must be set to one. If both bits 3 and 4 are reset to zero, the client will not receive artificial **CARD_INSERTION** notifications and also will not receive a **REGISTRATION_COMPLETE** notification.

Card Services passes the client's handle in the *Misc* argument of all **CARD_INSERTION** and **REGISTRATION_COMPLETE** event notifications. This eliminates the possibility of a race condition where Card Services begins making artificial **CARD_INSERTION** notifications before a client can record the client handle returned by this request.

I/O client device drivers are notified of card insertions before other clients. I/O clients are notified in LIFO order (i.e. the last I/O client to register is notified first) to ensure that the most recent, and presumed up to date, client is the one that sets the interface for the PC Card and socket. Memory Technology Drivers are notified of card insertions next in FIFO order (i.e. the first MTD to register is notified first). Finally, memory client device drivers are notified in FIFO order. Memory client device drivers are clients that use **the Open/ Close/ Read/ Write/ Copy/ EraseMemory** requests and so must be notified after the MTDs have associated with the memory regions that they will support. I/O clients must use the **RequestConfiguration** service to set the socket and card interface type before MTDs and memory clients begin accessing the card.

The *Event Mask* field is bit-mapped. Card Services performs event notification based on this field. The low-order eight bits specify events noted by Socket Services. The upper eight bits specify events generated by Card Services. The field is defined as follows:

Bit 0	Write Protect Change
Bit 1	Card Lock Change
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Dead
Bit 5	Battery Low
Bit 6	Ready Change
Bit 7	Card Detect Change
Bit 8	Power Management Change
Bit 9	Reset
Bit 10	Socket Services Updated
Bit 11	Extended Status Change
Bits 12 ·· 15	RESERVED (Reset to zero)

See the CARD_INSERTION callback section for additional information about handling events.

Setting bit 8 of the *Event Mask* indicates that the client is power management-aware and is able to handle all of the defined **PM_SUSPEND** and **PM_RESUME** events.

The *ClientData* field contains information that the client wants passed when its callback entry point is called. The contents of this field are binding specific. (See ***Appendix-D, 9. Bindings***.)

The *Version* field (in BCD) contains the specific Card Services *CSLevel* that the client expects to use. (See ***5.10 GetCardServicesInfo (0Bh)***.) This information can be used by Card Services to provide a better backward compatibility.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to fourteen (14)
BAD_ATTRIBUTE	No client type or more than one client type specified
BAD_VERSION	Card Services cannot support this version client
OUT_OF_RESOURCE	No space in Card Services to register client

5.38 RegisterEraseQueue (0FH)

CardServices(RegisterEraseQueue, *ClientHandle/EraseQueueHandle, EraseQueueHeader, 0, null*)

This service registers a client supplied erase queue with Card Services. The pointer to the *EraseQueueHeader* is passed in the Pointer argument. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument. The *EraseQueueHandle* for the Queue is returned in the *Handle* argument.

The *EraseQueueHeader* has the following structure:

Offset	Field	Size	Type	Value	Detail/Description
0	QueueEntryLen	2	I	N	Length in bytes of an erase queue entry.
2	QueueEntryCnt	2	I	N	Number of entries in the erase queue.
4	QueueEntryArray	N	I	N	Array of Erase Queue Entries.

The ***QueueEntryLen*** field specifies the size in bytes of each entry in the erase queue. The erase queue entries start at offset 4 from the beginning of the *EraseQueueHeader*, i.e. immediately after the *QueueEntryCnt* field of the erase queue *EraseQueueHeader*. The erase queue entries are contiguous and form an array of entries.

The *QueueEntryCnt* field specifies the number of entries in the erase queue.

Each *QueueEntry* has the following structure:

Offset	Field	Size	Type	Value	Detail/Description
0	Handle	2	I	N	*MemoryHandle*
2	EntryState	1	I/O	N	State of this erase queue entry
3	Size	1	I	N	Size of area to be erased (power of 2)
4	Offset	4	I	N	Offset of area to be erased
8	Optional	N	I	N	Additional bytes for client use

The *Handle* field contains the memory handle returned by an **OpenMemory** request for the memory area to be erased by this erase request.

The *EntryState* field indicates the state of this queue entry. The following states are defined:

State	Description
IDLE (FFH)	The erase queue entry has no valid information and should be ignored by Card Services. Should only be set when Card Services is not processing the request (as indicated by a value other than 01H - 7FH).
QUEUED_FOR_ERASE (00H)	The client has queued this entry for erasure. The client must not modify this entry until Card Services indicates that the erase has been processed. Only set by the Client before a **RegisterEraseQueue** or **CheckEraseQueue** request.
ERASE_IN_PROGRESS (01H - 7FH)	Card Services has started processing this entry. Only set by Card Services when the request is being serviced.
ERASE_PASSED (E0H)	Card Services has completed processing this entry and the erase was successful. The client can modify this entry. Only set by Card Services when the request has been serviced.
ERASE_FAILED (E1H)	Card Services has completed processing this entry and the erase was unsuccessful. The client can modify this entry. This entry code is only set by Card Services when the indicated block could not be erased after appropriate retries. The client is expected to treat the block indicated by this entry as unusable. Only set by Card Services when the request has been serviced.
MEDIA_WRITE_PROTECTED (84H) MEDIA_NOT_ERASABLE (86H) MEDIA_MISSING (80H) MEDIA_NOT_WRITABLE (87H)	Card Services has completed processing this entry and the erase was unsuccessful. The client can modify this entry. These codes indicate a user induced failure that can be corrected with appropriate user interaction. Only set by Card Services when the request cannot be serviced. MEDIA_NOT_ERASABLE is returned when an erase is attempted on an SRAM card supported by the default Card Services SRAM MTD.
BAD_SOCKET (C3H) BAD_TECHNOLOGY (C2H) BAD_OFFSET (C1H) BAD_VCC (C4H) BAD_VPP (C5H) BAD_SIZE (C6H)	Card Services has completed processing this entry and the erase was not attempted. The client can modify this entry. These codes indicate an error in the parameters of the entry. Only set by Card Services when the request cannot be serviced.

The *Size* field specifies the size of the memory area to be erased. It must be the exponent for a power of 2, e.g. a 64 KByte erase block request would specify a size of 16.

The *Offset* field specifies the offset to the memory area to be erased. It is a relative offset from the physical offset provided to **Open Memory** when the *MemoryHandle* was obtained. The offset must be the beginning of an erase block.

The *Optional* field is a byte array that can be used by the client and will not be accessed by Card Services.

All entries in an erase queue may be idle when registered. A return code of SUCCESS indicates the erase queue will be serviced by Card Services.

Return Codes

BAD_ARGS	QueueEntryCnt less than 1 or QueueEntryLen less than 8
BAD_ARG_LENGTH	*ArgLength* is not equal to zero (0)
BAD_HANDLE	*ClientHandle* is invalid

See also **DeregisterEraseQueue**.

5.39 RegisterMTD (1AH)

CardServices(RegisterMTD, *ClientHandle, null, ArgLength, ArgPointer*)

This service allows a Memory Technology Driver to register to handle accesses for a region by a memory service. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Attributes of memory type.
4	Offset	4	I	N	Card Offset for Region MTD supports
8	MTD Media ID	2	I	N	Token for MTD use to identify media

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. It indicates the type of memory that is being supported by the MTD as follows:

Bit 0	Memory type (set = attribute)
Bit 1	DeRegisterMTD
Bit 2	RESERVED (Reset to zero)
Bit 3 ·-4	Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use
Bits 5 ·· 7	RESERVED (Reset to zero)
Bits 8	RESERVED (Reset to zero)
Bits 9 ·· 10	Write/Erase interactions: 0 - Write without Erase 1 - Write with Erase 2 - Reserved 3 - Write with Disableable Erase
Bit 11	Write with Verify
Bit 12	Erase Requests Supported
Bits 13 ·· 15	Base Address Register number (1-7).

CardBus PC Cards do not have attribute memory, so *Memory Type* must always be reset.

DeRegisterMTD is set to one (1) when the MTD wishes to stop handling accesses for the indicated region.

Prefetchable / Cacheable applies to CardBus PC Cards only. 16-bit PC Cards shall use zero (0) for this field.

Write without Erase indicates no erase is done before a write. *Write with Erase* indicates writes that are erase block aligned and multiple erase block sized are erased before being written. *Write with Disableable Erase* indicates the *WriteMemory* attribute *DisableEraseBeforeWrite* can be used to control if an erase before write is not done. *Write with Verify* is set to one if writes can be verified after writing.

The *WriteMemory* attribute *Verify* is used to request a verified write. *Erase Requests Supported* indicates that erase requests via an *EraseQueue* are supported for this partition.

The *Offset* field identifies the memory region for which the MTD supports access. This value must be the beginning address of a region.

The *MTD Media ID* is a value that Card Services maintains on behalf of an MTD. The MTD uses this value to indicate the type of memory media for this region. This value is passed to the MTD by Card Services whenever a read, write, or erase memory access service is requested for this region.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to ten (10)
BAD_HANDLE	*ClientHandle* is invalid
BAD_OFFSET	Offset is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
NO_CARD	No PC Card in socket

5.40 RegisterTimer (28H)

CardServices(RegisterTimer, *ClientHandle/TimerHandle, null, ArgLength, ArgPointer*)

This service 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 *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument. A *TimerHandle* is returned in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Wait	2	I	N	Number of ticks to wait

The *Wait* field is the number of timer intervals Card Services should wait before notifying the client. The tick interval is approximately 1 ms. See warning below. If the *Wait* field is zero (0) on entry, Card Services notifies the client as soon as the Card Services interface is present.

This service is intended for use by two types of clients. First, those who may be operating in a background thread of execution. Second, clients who need periodic wakeup calls.

Background thread of execution clients may require Card Services to perform a task when the Card Services interface is busy with a foreground request. Since this service is always available, even when the Card Services interface reports BUSY on all other requests, it allows a client to schedule a later wakeup when Card Services is available to perform the delayed request.

This service may also serve as a method of receiving periodic wakeup notifications without a client having to intercept the system timer tick. As noted above, a side effect is that when the client receives notification the timer has expired, the Card Services interface is guaranteed to be available.

When the timer expires and the Card Services interface is available, Card Services notifies the client at the address specified when the client was registered with **RegisterClient**. Clients can use the *TimerHandle* passed to their callback routine to determine which timer has expired.

A client may have more than one timer pending at a time. Card Services returns OUT_OF_RESOURCE if it cannot accept the timer request.

WARNING:

When Card Services is operating in some environments, timer intervals may not be received by Card Services reliably. Many environments (such as Microsoft Windows or systems running LIM emulators on Virtual-86 capable processors) may generate timer intervals in bursts. The actual interval available on a system may not be as little as 1 ms. This interval should only be viewed as advisory. Other Operating System specific services should be used for reliable timing.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
BAD_HANDLE	*ClientHandle* is invalid
OUT_OF_RESOURCE	Timer request invalid

5.41 ReleaseConfiguration (1EH)

CardServices(ReleaseConfiguration, *ClientHandle, null, ArgLength, ArgPointer*)

The **ReleaseConfiguration** and corresponding **RequestConfiguration** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service returns a 16-bit PC Card and its socket to a simple memory only interface and configuration zero. For CardBus PC Cards this disables I/O accesses to the function.

Card Services may remove power from the socket if no clients have indicated their usage of the socket by an **OpenMemory** or **RequestWindow**. The *ClientHandle* used in **RequestConfiguration** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

BAD_HANDLE is returned if the *ClientHandle* is not the one passed to **RequestConfiguration**.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
BAD_HANDLE	*ClientHandle* does not match owning client or no configuration to release
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)

5.42 ReleaseExclusive (2DH)

CardServices(ReleaseExclusive, *ClientHandle, null, ArgLength, ArgPointer*)

This service releases the exclusive use of a card in a socket for a client. The *ClientHandle* passed to **RequestExclusive** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit Mapped field.

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bits 0 ·· 15	RESERVED (Reset to zero)

Card Services returns to the client immediately. As soon as the Card Services interface is available, Card Services sends a **CARD_REMOVAL** event to the requesting client and then sends a **CARD_INSERTION** event to all registered clients that have requested insertion events.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_HANDLE	*ClientHandle* does not match owning client or no client has exclusive use of the PC Card in the socket
BAD_SOCKET	*Socket* or function is invalid

See also **RequestExclusive**.

5.43 ReleaseIO (1BH) [16-bit PC Card only]

CardServices(ReleaseIO, *ClientHandle, null, ArgLength, ArgPointer*)

The **ReleaseIO** and corresponding **RequestIO** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service releases I/O addresses requested with the **RequestIO** service. Only the Card Services database of resources is adjusted by this service. No changes are made to the socket adapter. **ReleaseIO** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket without a matching **ReleaseConfiguration**. The *ClientHandle* used in **RequestIO** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Base Port1	2	I	N	Base port address for range 1
4	Num Ports1	1	I	N	Number of contiguous ports
5	Attributes1	1	I	N	Bit-mapped
6	Base Port2	2	I	N	Base port address for range 2
8	Num Ports2	1	I	N	Number of contiguous ports
9	Attributes2	1	I	N	Bit-mapped
10	IOAddrLines	1	I	N	Number of I/O address lines decoded by a 16-bit PC Card. For a CardBus PC Card, this is ignored.

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Base Port* fields describe the first port address assigned by a **RequestIO** request. It must match the value returned by the **RequestIO** service exactly.

The *Num Ports* fields describe the number of contiguous ports assigned by a **RequestIO** service.

The *Attributes* fields are defined the same as for **RequestIO** and must have the same value returned by **RequestIO**.

For 16-bit PC Cards the *IOAddrLines* field is the number of I/O address lines decoded by the PC Card in the specified socket. It is used by Card Services to determine whether any addresses outside the ranges specified needed to be marked as in-use because the combination of socket hardware and lines decoded could result in PC Card accesses outside the specified ranges. For CardBus PC Cards, this is ignored.

Releasing ports using different *Base Port*, *Num Ports* or *Attributes* values then those used by the corresponding **RequestIO** is not supported. Doing so may crash the host system or confuse resource allocation.

SUCCESS is returned if the specified ports were in use and have been released.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eleven (11)
BAD_ARGS	I/O description doesn't match allocation
BAD_HANDLE	*ClientHandle* does not match owning client or no I/O ports to release.
BAD_SOCKET	*Socket* or function is invalid
CONFIGURATION_LOCKED	Configuration has not been released

5.44 ReleaseIRQ (1CH)

CardServices(ReleaseIRQ, *ClientHandle, null, ArgLength, ArgPointer*)

The **ReleaseIRQ** and corresponding **RequestIRQ** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service releases a previously requested interrupt request line. Only the Card Services database of resources is adjusted by this service. No changes are made to the socket adapter. **ReleaseIRQ** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket without a matching **ReleaseConfiguration**. The *ClientHandle* used in **RequestIRQ** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped
4	AssignedIRQ	1	I	N	IRQ Number Assigned by CS

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped and is defined the same as in **RequestIRQ**. The *AssignedIRQ* field identifies the IRQ that was previously established by **RequestIRQ**.

Note: Most systems have hardware support for interrupt reporting and an interrupt handler vector (jump) table. This service does not manipulate any such motherboard specific hardware nor does it manipulate the vector table. It is up to the client to perform these activities, if the IRQ is not shared before invoking this service. No adjustment of the interrupt vectors in the interrupt table is made by this request. It is up to the client to restore the original interrupt handler after invoking this service.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to five (5)
BAD_ATTRIBUTE	*Attributes* don't match allocation
BAD_IRQ	*AssignedIRQ* doesn't match allocation
BAD_HANDLE	*ClientHandle* does not match owning client or no IRQ to release.
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
CONFIGURATION_LOCKED	Configuration has not been released

5.45 ReleaseSocketMask (2FH)

CardServices(ReleaseSocketMask, *ClientHandle, null, ArgLength, ArgPointer*)

This service requests that the client no longer be notified of status changes for this socket. A client will still be notified of status changes for this socket if it has events enabled in its global event mask. The *ClientHandle* passed to **RequestSocketMask** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
BAD_HANDLE	*ClientHandle* does not match owning client or no socket mask to release
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)

5.46 ReleaseWindow (1DH)

CardServices(ReleaseWindow, *WindowHandle, null, 0, null*)

The **ReleaseWindow** and corresponding **RequestWindow** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service releases a block of system address space which was obtained previously by a corresponding **RequestWindow**. The *WindowHandle* returned by **RequestWindow** is passed in the *Handle* argument.

Card Services assumes only the owning client will make this request. For that reason, the owning client handle is not provided as an argument. Card Services maintains the owning client's handle in its internal database to properly manage resources.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to zero (0)
BAD_HANDLE	*WindowHandle* is invalid

5.47 ReplaceSocketServices (33H)

CardServices(ReplaceSocketServices, *null, SSEntry, ArgLength, ArgPointer*)

This service allows a new Socket Services handler to replace an existing one that Card Services is using or for an existing Socket Services to change the count of sockets that it supports. For the case of a replacement the new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The pointer to the Socket Services entry point is passed in the Pointer argument. Card Services calls Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	N	Information about SS entry point
6	DataPointer	N	I	N	Pointer for SS Data Area (binding specific)

The *Socket* field indicates the logical socket that is the first socket controlled by the Socket Services handler to be replaced. The *NumSockets* field indicates the number of sockets that are controlled by that Socket Services handler. If these values do not describe an installed Socket Services handler, this service fails.

The *Attributes* field defines details about the new Socket Services entry point. The definition is binding specific.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in a binding specific way. This field is defined the same as other (binding specific) pointers.

BAD_SOCKET is returned if a single existing Socket Services handler does not control all of the indicated sockets.

In the case of a socket count change if a Card Services implementation is utilizing the external chaining method (e.g. Socket Services **GetSetPriorHandler** service) for tracking Socket Services handlers then Card Services needs to remove the requesting handler from the chain before returning from this request. The method that Card Services removes the handler is by issuing a series of **GetSetPriorHandler** requests that effectively remove the requesting handler from the chain of handlers. At this point Card Services can return from this request and continue processing from a background thread.

Note: Several methods are possible for a Card Services implementation to track Socket Services handlers. Depending upon the implementation Card Services may not issue any **GetSetPriorHandler** requests. Socket Services implementations should be able to handle this situation.

In the background thread Card Services will issue SS_UPDATED event callbacks to all clients with SocketsRemoved as the New Sockets parameter. This will be followed by determining if any socket renumbering is necessary. If renumbering is determined to be required then Card Services will issue to all clients SS_UPDATED event callbacks with SocketRenumber as the New Sockets parameter for any affected sockets. Next Card Services adds the original requesting Socket Services handler to the end of the “chain” of handlers. Lastly, Card Services notifies all clients parameter for any sockets that

were added by the requesting Socket Services handler an SS_UPDATED callback event with SocketsAdded as the New Sockets.

Note: Card Services implementations may choose not to renumber sockets in which case clients will not receive any SS_UPDATED event callbacks with the New Sockets parameter set to SocketRenumber.

Return Codes

BAD_ARG_LENGTH	*ArgLength* value invalid - mode dependent
BAD_ARGS	NumSockets invalid
BAD_SOCKET	*Socket* is invalid
UNSUPPORTED_MODE	Requested processor mode not supported

5.48 RequestConfiguration (30H)

CardServices(RequestConfiguration, *ClientHandle, null, ArgLength, ArgPointer*)

The **RequestConfiguration** and corresponding **ReleaseConfiguration** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service configures the PC Card and socket. Card Services applies power to the socket if the socket was not powered. This service must be used by clients that require I/O windows to be enabled or **IREQ#** routing. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument.

All I/O windows and **IREQ#** routing must have been requested before this service is used. I/O windows are requested using the **RequestIO** service or the **RequestWindow** service. **IREQ#** routing is requested using the **RequestIRQ** service. (See ***5.50 RequestIO (1Fh)*** [16-bit PC Card only] and ***5.53 RequestWindow (21h)***.)

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	**VCC**	1	I	N	**VCC** Setting
5	**VPP1**	1	I	N	**VPP1** Setting
6	**VPP2**	1	I	N	**VPP2** Setting
7	IntType	1	I	N	Memory-Only, Memory and I/O Interface, or Custom Interface
8	ConfigBase	4	I	N	Card Base address of config registers
12	Status	1	I	N	Card Status register setting, if present
13	Pin	1	I	N	Card Pin register setting, if present
14	Copy	1	I	N	Card Socket/Copy register setting, if present
15	ConfigIndex	1	I	N	Card Option register setting, if present
16	Present	1	I	N	Card Configuration registers present
17	Extended Status	1	I	N	Extended Status Register Setting (if present)
18	Custom Interface ID Number	4	I	N	Custom Interface ID Number (if IntType set to Custom Interface).

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The routing of the IRQ signals may be enabled or left disabled, if these resources were previously acquired with **RequestIRQ**. BAD_ATTRIBUTE is returned if an attempt is made to enable IRQ steering when these resources were not previously acquired. The following bits are defined:

Bit 0	RESERVED (Reset to zero)
Bit 1	Enable IRQ steering (set = true)
Bits 2 ·· 5	RESERVED (Reset to zero)
Bit 6	RESERVED[1] (Reset to zero)
Bits 7 ·· 8	RESERVED (Reset to zero)
Bit 9	VSOVERRIDE (set = override VS pins)
Bits 10 ·· 15	RESERVED (Reset to zero)

1. Legacy feature no longer supported.

After card insertion and prior to the first successful **RequestConfiguration**, the voltage levels applied to the card shall be those indicated by the card's physical key and / or the **VS[2::1]** voltage sense pins. (See the ***Electrical Specification***.) For Low Voltage capable host systems (hosts which are capable of VS pin decoding), if a client desires to apply a voltage not indicated by the VS pin decoding then the VSOVERRIDE bit must be set in the *Attribute* field otherwise a BAD_VCC shall be returned. The 5.0 volt level is never a valid **VCC** setting for CardBus PC Cards.

PC Cards indicate multiple **VCC** voltage capability in their CIS, see the Metaformat Specification for details. After card insertion, Card Services processes the CIS, and when multiple **VCC** voltage capability is indicated, Card Services will allow the client to apply **VCC** voltage levels which are contrary to the VS pin decoding without setting the VSOVERRIDE bit.

WARNING:

The VSOVERRIDE bit is provided for clients that have a need to override the information provided in the CIS. The Client must exercise caution when setting this bit as it overrides any voltage level protection provided by Card Services.

Setting Bit 1 to one enables the IRQ steering as requested by **RequestIRQ**. Resetting Bit 1 to zero disables the IRQ steering.

The **VCC**, **VPP1** and **VPP2** fields all represent voltages expressed in tenths of a volt. Since these fields are a byte wide, values from zero (0) to 25.5 volts may be set. To be valid, the exact voltage must be available through the system's Socket Services.

All functions on a PC Card must support the same voltage or voltages. **VCC** is set to the highest **VCC** voltage requested for any function. **VPP[2::1]** is set to the highest **VPP1** or **VPP2** voltage requested for any function.

For 16-bit PC Cards, the *ConfigBase* field is the offset in attribute memory of the configuration registers. For CardBus PC Cards, *ConfigBase* has the same format as the CIS Pointer in CardBus PC Card configuration space which is also that of the *TPCC_ADDR* field of the *CISTPL_CONFIG_CB* tuple as well as the for the *TPLL_ADDR* field in the CISTPL_LONGLINK_CB tuple. The *Present* field identifies which, if any, of the configuration registers are present. If present, the corresponding bit is set. For CardBus PC Cards, all four (4) CardBus PC Card status registers are always present. However, these registers have different definitions than those for 16-bit PC Cards and so the first five (5) bits of this field will always be reset to zero (0) for CardBus PC Cards. This field is bit-mapped as follows:

Bit 0	Option
Bit 1	Status
Bit 2	Pin Replacement
Bit 3	Copy
Bit 4	Extended Status
Bit 5	I/O Base 0
Bit 6	I/O Base 1
Bit 7	I/O Base 2

If the *Present* field indicates that any I/O Base registers are present, Card Services writes values (corresponding to the single I/O range requested via either the Request IO or the Request Window service) to all I/O Base registers and the I/O Limit register.

The *IntType* field is a set of mutually exclusive flags. It indicates how the socket should be configured. The IF_CARDBUS flag shall only be set for CardBus PC Cards. The IF_CUSTOM flag is set when a custom interface is selected and a Custom Interface ID Number is specified. The following bits are defined:

Bit 0	Memory (set = true)
Bit 1	Memory and I/O (set = true)
Bit 2	IF_CARDBUS (set = true)
Bit 3	IF_CUSTOM
Bits 4 ·· 7	RESERVED (Reset to zero).

The *Custom Interface ID Number* field is used when the IF_CUSTOM interface type is selected in the *IntType* field. This Interface Number is a PCMCIA and JEITA jointly assigned value that identifies a specific custom interface. To be valid, the *custom interface indicated by the ID* Number must be available through the system's Socket Services. (See also the discussion of Custom Interface Subtuples under CISTPL_CONFIG in the ***Metaformat Specification***.)

Only one client can be in control of the interface type at any time. Once an interface type other than Memory has been set, a **ReleaseConfiguration** must be used to return the socket to a memory only interface before another I/O or custom interface configuration can be selected.

For 16-bit PC Cards, the *Status*, *Pin*, *Copy*, and *Extended Status* fields represent the initial values that should be written to those registers if they are present, as indicated by the *Present* field. The *Pin* field is also used to inform Card Services which pins in the PC Card's Pin Replacement Register are valid, if any. Only those bits which are set are considered valid. This affects how status is returned by the **GetStatus** service. If a particular signal is valid in the Pin Replacement Register, both the mask bit and the change bit must be set in the *Pin* field.

Multiple Function 16-bit PC Cards Base and Limit registers are set by the **RequestConfiguration** service.

The *ConfigIndex* field is the value written to the Option register for the configuration index required by the PC Card. Only the least significant six bits are significant, the upper two (2) bits are ignored. The interrupt type is set by Card Services based on the client's prior **RequestIRQ** request and the host hardware environment.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to seventeen (17), eighteen (18), twenty-two (22) or *ArgLength* is not equal to twenty-two (22) and *IntType* field = IF_CUSTOM
BAD_ATTRIBUTE	IRQ steering enable conflict
BAD_HANDLE	*ClientHandle* is invalid
BAD_TYPE	I/O and memory interface is not supported or the specific custom Interface ID Number specified is not supported
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
BAD_VCC	Requested voltage is not available on socket or other function has conflict
BAD_VPP	Requested voltage is not available on socket or other function has conflict
CONFIGURATION_LOCKED	Configuration already set
NO_CARD	No PC Card in socket
IN_USE	PC Card already being used

5.49 RequestExclusive (2CH)

CardServices(RequestExclusive, *ClientHandle, null, ArgLength, ArgPointer*)

This service requests the exclusive use of a PC Card in a socket for a client. Clients currently using the PC Card in the socket can reject the request. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument. This service returns without indicating whether the client received exclusive access to the PC Card. The client is notified via its callback entry point whether it has received exclusive access. This notification can happen before or after this service returns.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped field (defined below)

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bits 0 ·· 15	RESERVED (Reset to zero)

Card Services returns to the client immediately after noting the request . As soon as the Card Services interface is available, Card Services sends **EXCLUSIVE_REQUEST** events to registered clients for this socket. If any client returns failure for the **EXCLUSIVE_REQUEST** event, Card Services terminates notification processing. Card Services then notifies the requesting client that the exclusive request failed via an **EXCLUSIVE_COMPLETE** event with the Info argument set to the return code set by the client that rejected the request.

Once all clients have accepted the **EXCLUSIVE_REQUEST** event, Card Services sends **CARD_REMOVAL** events to all clients registered and then sends a **CARD_INSERTION** event to the requesting client. Finally, Card Services sends the **EXCLUSIVE_COMPLETE** event to the requesting client.

When the client is through using the PC Card in an exclusive fashion, it must use the **ReleaseExclusive** service to return the PC Card to normal use.

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to four (4)
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
IN_USE	PC Card already in-use exclusively or **RequestExclusive** already in process
NO_CARD	No PC Card in socket

5.50 RequestIO (1FH) [16-bit PC Card only]

CardServices(RequestIO, *ClientHandle, null, ArgLength, ArgPointer*)

The **RequestIO** and corresponding **ReleaseIO** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

With February 1995 publication of the ***Card Services Specification***, I/O address ranges are also requested using the enhanced **RequestWindow** service. CardBus PC Cards require the **RequestWindow** service to allocate I/O address ranges. 16-bit PC Card I/O address ranges may also be specified using the enhanced **RequestWindow** service. (See ***5.53 RequestWindow (21h)***.)

The **RequestIO** service determines if host system I/O address ranges may be allocated to a 16-bit PC Card. Card Services confirms that the resource is available and that the socket controller is capable of performing the routing(s). If this service is successful, the host system I/O address ranges are reserved. Access is not enabled until the **RequestConfiguration** service has been invoked successfully.

When I/O address space is no longer required, it must be released using the **ReleaseIO** service. If the **RequestConfiguration** service has been successfully invoked, the hardware configuration must first be released using the **ReleaseConfiguration** service.

The **RequestConfiguration** service only locks the last unreleased I/O address space(s) specified by a successful **RequestIO** service invocation. If a particular combination of I/O range(s) and interrupt routing is required, **RequestIO** and **ReleaseIO** service requests may be interleaved with **RequestIRQ** and **ReleaseIRQ** services until an acceptable combination is achieved. Socket and PC Card hardware is not programmed for the selected configuration until the **RequestConfiguration** service is requested and completes successfully.

The client must provide its assigned *ClientHandle* in the *Handle* argument. This is the value returned when the Client registered with Card Services using the **RegisterClient** service.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Base Port1	2	I/O	N	Base port address for range
4	Num Ports1	1	I	N	Number of contiguous ports
5	Attributes1	1	I	N	Bit-mapped
6	Base Port2	2	I	N	Base port address for range
8	Num Ports2	1	I	N	Number of contiguous ports
9	Attributes2	1	I	N	Bit-mapped
10	IOAddrLines	1	I	N	Number of I/O address lines decoded by a 16-bit PC Card. For a CardBus PC Card, this is ignored

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

Two I/O address ranges can be requested by **RequestIO**. Each I/O address range is specified by the *Base Port*, *Num Ports*, and *Attributes* fields. If only a single I/O range is being requested, the *Num Ports2* field must be reset to zero.

The *Base Port* field specifies the base port address requested. If reset to zero (0) Card Services returns an I/O address based on the available I/O addresses and the number of contiguous ports requested. Under these circumstances, Card Services aligns the returned range in the host system's I/O address space on a boundary that is a multiple of the number of contiguous ports requested rounded up to the nearest power of two. For example, if a client requests two I/O ports, the returned *Base Port* value will be two. If a client requests five contiguous I/O ports, the returned *Base Port* value will be eight. *If multiple ranges are being requested,* the ***Base Port*** field must be non-zero for all specified ranges.

The *Num Ports* field is the number of contiguous ports being requested.

The *Attribute* fields are bit-mapped. The following bits are defined:

Bit 0	Shared (set = true)
Bit 1	First Shared (set = true)
Bit 2	Force Alias Accessibility
Bit 3	Data Path Width for I/O Range 0 = 8 bit 1 = 16 bit
Bits 4 ·· 7	Reserved (must be reset to zero)

Normally, each request dedicates the requested ports to the indicated socket, if they are available. However, for some applications and/or PC Cards, ports may be shared by cards in two or more sockets. If the *Shared* bit is set, the ports requested may be shared with another socket. *First Shared* is additionally set when a previously unshared I/O range is required that is intended to be shared with subsequent clients using this same I/O range. If a previously unshared I/O range is unavailable, the service fails and IN_USE is returned.

If a shared I/O range is requested, the client is responsible for determining whether the range may be shared.

A PC Card may decode less than the full set of possible I/O address lines. Doing so creates aliased addresses for the PC Card address range by using different values for the undecoded upper address lines. *Force Alias Accessibility* requests that the aliased address ranges be configured so that they can also be used to address the PC Card. This is used for compatibility with similar functionality on existing ISA bus style add-in cards. If Card Services can only satisfy this request by aliasing the requested I/O addresses with other I/O addresses and *Force Alias* is not set to one, the BAD_ATTRIBUTE error will be returned.

Note: This capability may not be available for all systems and clients should not depend on being able to request an address range with alias accessibility.

The data path width is specified by Bit 3 of the Attributes field.

Shared ports are managed internally by Card Services with share counts. Once internal share counts reach zero (0), those ports may be reassigned for exclusive use. If share counts for I/O ports are non-zero, they may only be shared if requested by **RequestIO**. If socket hardware does not support shared I/O ports for a shared request or is unable to satisfy a non-shared request, this service returns failure.

The *IOAddrLines* field is the number of I/O address lines decoded by the PC Card in the specified socket. It is used by Card Services to determine whether any addresses outside the ranges specified needed to be marked as in-use because the combination of socket hardware and lines decoded could result in PC Card accesses outside the specified ranges.

SUCCESS is returned if the specified ports are available.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eleven (11)
BAD_ATTRIBUTE	Sharing or alias request invalid or CardBus PC Card is inserted in specified socket
BAD_BASE	Base port address is invalid
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
CONFIGURATION_LOCKED	Configuration already set
IN_USE	I/O ports requested are already in-use or service has already been successfully invoked
NO_CARD	No PC Card in socket
OUT_OF_RESOURCE	Internal data space exhausted

5.51 RequestIRQ (20H)

CardServices(RequestIRQ, *ClientHandle, ISRAddress | null, ArgLength, ArgPointer*)

The **RequestIRQ** and corresponding **ReleaseIRQ** services are no longer recommended for use as part of the process of configuring a function on a PC Card for use. Instead, the **InquireConfiguration** and **ConfigureFunction** services provide a more efficient means of configuration.

This service determines if a PC Card's interrupt signal may be routed to a host system interrupt request line. Card Services confirms that the host system interrupt is available for the requested level of service (exclusive, shared, etceteras) and that the socket controller is capable of performing the routing. If this service is successful, the host system interrupt request routing is reserved for later assignment using the **RequestConfiguration** service. Hardware routing is not performed until the **RequestConfiguration** service has been invoked successfully.

When the routing is no longer required, it must be released using the **ReleaseIRQ** service. If the **RequestConfiguration** service has been successfully invoked, the hardware configuration must first be released using the **ReleaseConfiguration** service.

The **RequestConfiguration** service only locks the last unreleased interrupt routing specified by a successful **RequestIRQ** service invocation. If a particular combination of I/O range(s) and interrupt routing is required, **RequestIRQ** and **ReleaseIRQ** service requests may be interleaved with **RequestIO** and **ReleaseIO** or **RequestWindow** and **ReleaseWindow** services until an acceptable combination is achieved. Socket and PC Card hardware is not programmed for the selected configuration until the **RequestConfiguration** service is requested and completes successfully.

The client must provide its assigned *ClientHandle* in the *Handle* argument. This is the value returned when the Client registered with Card Services using the **RegisterClient** service.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Bit-mapped (defined below)
4	AssignedIRQ	1	O	N	IRQ Number Assigned by CS
5	IRQInfo1	1	I	N	First TPCE_IR IRQ Byte
6	IRQInfo2	2	I	N	Optional TPCE_IR IRQ bytes

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. It specifies details about the type of IRQ desired by the client.

The following bits are defined in the Attributes field:

Bits 0 ·· 1	IRQ type: 0 - Exclusive 1 - Time-Multiplexed Sharing 2 - Dynamic Sharing 3 - RESERVED
Bit 2	Force Pulse (set = true) (Ignored for CardBus PC Card)
Bit 3	First Shared (set = true)
Bit 4	*ISRAddressProvided* (set = Pointer argument contains binding specific address of interrupt service routine)
Bit 5 ·· 7	RESERVED (Reset to zero)
Bit 8	Pulse IRQ Allocated (set = true on return) (Not used for CardBus PC Card)
Bits 9 ·· 14	RESERVED (Reset to zero)
Bit 15	This bit is used by the **InquireConfiguration** and **ConfigureFunction** services.

The *IRQ type* field specifies the characteristics of the IRQ requested by the client. Exclusive indicates that the system IRQ is dedicated to this PC Card.

Time-Multiplexed Sharing indicates the client shares the system IRQ with other PC Cards. This client coordinates with other clients in using **ModifyConfiguration** to enable / disable the IRQ from each socket. This ensures that only one PC Card is connected to the system IRQ line at any time. A time-multiplexed IRQ is only supported for interrupts that can be enabled and disabled at the socket.

Dynamic Sharing indicates that this PC Card will share the system IRQ simultaneously with other PC Cards. The clients of the PC Cards use card features to identify the source of the interrupt. Dynamic sharing is available via PC Card level interrupts in systems that support level mode interrupts. Dynamic sharing is also available via PC Card pulse interrupts in systems that support pulse mode interrupts. Level mode interrupts are assigned if possible. *Force Pulse* is used to force Card Services to use a pulse mode interrupt.

First Shared is set when a previously unshared IRQ is required that is intended to be shared with subsequent clients using this same IRQ. If a previously unshared IRQ is unavailable, the service fails and BAD_IRQ is returned. *First Shared* is only valid when *Time Multiplexed Sharing* or *Dynamic Sharing* is specified for the IRQ type.

Clients specify the address of a routine to handle interrupt events by setting the *ISRAddressProvided* bit in the *Attributes* field to one and providing a binding specific pointer to their routine in the *Pointer* argument. Card Services installs a First-Level Interrupt Handler (FLIH) on the assigned interrupt vector that initially receives all interrupt notifications from the PC Card. Control is routed to Client handlers using a CALL instruction. On entry to the client handler the FLIH has preserved all registers and provided one hundred twenty-eight (128) words of stack space. A client routine requiring more stack space than this shall provide its own suitably sized stack space. When function specific interrupt processing is complete, the Client handler returns control to Card Services using a RET instruction. The handler shall indicate either that an interrupt condition was serviced by returning with the CARRY flag set or that the function did not require interrupt service by returning with the CARRY flag clear.

When Clients set the *ISRAddressProvided* field to one (1), the Card Services FLIH is responsible for performing End-Of-Interrupt (EOI) processing, acknowledging the completion of interrupt processing

to the PC Card (if required) and returning from the PC Card's interrupt notification. Also, when the *ISRAddressProvided* field is set to one (1), Card Services ignores those **RequestIRQ** parameters that specify details about the type of IRQ desired by the client. These parameters include the *IRQInfo1* and *IRQInfo2* fields as well as the *IRQ type* (bits 0 and 1), *Force Pulse* (bit 2) and *Pulse IRQ Allocated* (bit 8) fields in the *Attributes* field. When **RequestIRQ** is successfully invoked with the *ISRAddressProvided* field set to one (1), the value returned in the *AssignedIRQ* field will be FEH. The *AssignedIRQ* field value of FEH indicates that a private IRQ has been reserved for or is in use by a FLIH and that the client has no need to know the actual interrupt level being used.

If the *ISRAddressProvided* field is reset to zero (0), Card Services does not invoke the Client's interrupt handler using a CALL instruction. Instead, the Client installs their interrupt handler on the assigned interrupt vector. In this case, the Client's interrupt handler is responsible for EOI processing.

Only one client may use the **RequestIRQ** service for a particular logical socket with the *ISRAddressProvided* field reset to zero (0). If there are other clients using interrupt notifications from other functions on the PC Card, they must provide an *ISRAddress* for their interrupt handler as specified above.

When a PC Card with multiple functions has multiple clients requesting interrupt notifications, the FLIH is responsible for dispatching the hardware notification to the appropriate interrupt handlers. As noted above, the handlers for Clients which provided an *ISRAddress* are invoked using a CALL instruction. If a Client has requested interrupt routing without specifying an *ISRAddress* (the *ISRAddressProvided* field reset to zero), this Client's interrupt handler is invoked by simulating a hardware interrupt on a different vector than the one actually used to receive hardware interrupt notifications from the PC Card.

When both types of client interrupt handlers are present, the Card Services FLIH is responsible for resolving what EOI processing is performed by the handler invoked by the simulated interrupt and any additional EOI processing required based on the interrupt level simulated and the interrupt level actually used by the PC Card. The Card Services FLIH is always responsible for signaling the completion of interrupt handling to a multiple function PC Card.

Card Services FLIH support for EOI processing and signaling the completion of interrupt handling is only available for PC Cards that follow the PC Card Standard definition of multiple function cards. This includes 16-bit PC Cards with multiple sets of configuration registers and CardBus PC Cards with multiple configuration spaces.

PC Cards that provide multiple functions using vendor-specific implementations (for example, 16-bit PC Cards with a single set of configuration registers) are responsible for distributing the single interrupt notification from the card to all interested clients, performing EOI processing and notifying the card the interrupt has been processed. For such cards, a single client typically uses the **RequestIRQ** service once with the *ISRAddressProvided* field reset to zero (0).

Pulse IRQ Allocated is valid only for dynamic shared IRQs. It indicates whether the PC Card will be configured for pulse or level operation. If a shared IRQ is requested, the *ClientHandle* of the requesting client must be used for this request.

Card Services maintains internal share counts for returned IRQ values that indicate sharing. These counts are incremented as IRQ are assigned and decremented when released. If an IRQ internal share count is zero, the IRQ may be exclusively assigned to a socket.

The *AssignedIRQ* field is returned by Card Services if one of the requested IRQ levels is available and was assigned to the socket. It is a value from zero (0) to nineteen (19) or, when the *ISRAddressProvided* field is set to one (1), the value FEH. Zero (0) through fifteen (15) correspond to IRQ levels 0 through 15, respectively.

The *IRQInfo1* field is bit-mapped. It indicates the capabilities of the PC Card in the socket. Its structure is identical to the Configuration Table Entry Tuple *TPCE_IR* field first Interrupt Description IRQ byte. The structure is:

if Bit 4 is reset to zero:

Bits 0 ·· 3	IRQN if Bit 4 is reset to zero

if Bit 4 is set to one:

Bit 0	NMI
Bit 1	IOCK
Bit 2	BERR
Bit 3	VEND

Bit 4	Mask, if set to one, the IRQInfo2 bytes are valid

Bit 5	Level
Bit 6	Pulse
Bit 7	Share

The *IRQInfo2* field is bit-mapped. These two bytes have identical structure to the Configuration Table Entry Tuple *TPCE_IR* field Interrupt Description optional IRQ bytes. The structure is:

Bit 0	IRQ0
Bit 1	IRQ1
Bit 2	IRQ2
Bit 3	IRQ3
Bit 4	IRQ4
Bit 5	IRQ5
Bit 6	IRQ6
Bit 7	IRQ7

Bit 8	IRQ8
Bit 9	IRQ9
Bit 10	IRQ10
Bit 11	IRQ11
Bit 12	IRQ12
Bit 13	IRQ13
Bit 14	IRQ14
Bit 15	IRQ15

Multiple bits may be set in the *IRQInfo1* and *IRQInfo2* fields. At least one bit must be set in these fields. How Card Services selects one level over another when more than one level is currently available is implementation-specific.

Note: Most systems have hardware support for interrupt reporting and an interrupt handler vector (jump) table. This service does not manipulate any such motherboard specific hardware nor does it manipulate the vector table when the *ISRAddressProvided* field is reset to zero (0). It is up to the client to perform these activities, if the IRQ is not shared before invoking this service.

Clients MUST ensure that the PC Card does not generate an interrupt before the interrupt handler is installed by the client on the appropriate interrupt vector.

SUCCESS is returned if one of the specified IRQs is available with the sharing requested. BAD_IRQ is returned if no IRQ satisfying the request can be found.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to eight (8)
BAD_ARGS	IRQ Info fields are invalid
BAD_ATTRIBUTE	Sharing or pulse request invalid
BAD_HANDLE	*ClientHandle* is invalid
BAD_IRQ	IRQ is invalid (returned by Socket Services)
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
CONFIGURATION_LOCKED	The configuration has been locked by the **RequestConfiguration** service
IN_USE	IRQ requested is already in-use or service has already been successfully invoked
NO_CARD	No PC Card in socket

5.52 RequestSocketMask (22H)

CardServices(RequestSocketMask, *ClientHandle, null, ArgLength, ArgPointer*)

This service requests that the client be notified of status changes for this socket. If the client also has events enabled in its global event mask, it may be notified more than once for each status change for this socket. The *ClientHandle* is passed in the *Handle* argument. **RequestSocketMask** must be used before a **Get/SetEventMask** request for this socket will succeed.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	EventMask	2	I	N	Bit-mapped (defined below)

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Event Mask* is a bit-mapped field. It represents the status callback events to notify the client of when they occur on this socket. The bits are defined as follows:

Bit 0	Write Protect
Bit 1	Card Lock Change
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Dead
Bit 5	Battery Low
Bit 6	Ready Change
Bit 7	Card Detect Change
Bit 8	PM Change
Bit 9	Reset
Bit 10	SS Update
Bit 11	Extended Status Change
Bits 12 ·· 15	RESERVED (Reset to zero)

BAD_SOCKET is returned if the *Socket* field is invalid.

Note: Socket event masks must be released when a **CARD_REMOVAL** notification is received.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
IN_USE	PC Card **RequestSocketMask** already in process
NO_CARD	No PC Card in socket

5.53 RequestWindow (21H)

CardServices(RequestWindow, *ClientHandle/WindowHandle, null, ArgLength, ArgPointer*)

This service requests a block of system address space be assigned to a PC Card in a socket. The *ClientHandle* of the requesting client is passed in the *Handle* argument. The *WindowHandle* is returned in the *Handle* argument. This *WindowHandle* must be passed to **ReleaseWindow** when the client is done using the window.

When a request is made for I/O resources, the **RequestWindow** service determines if the host system I/O address ranges may be allocated. If this service is successful, the host system I/O address ranges are reserved. Access to the I/O range is not enabled until the **RequestConfiguration** service has been invoked successfully.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Memory Window Attribute Field
4	Base	4	I/O	N	System Base Address
8	Size	4	I/O	N	Memory Window Size
12	AccessSpeed or IOAddrLines	1	I	N	Window Speed Field or number of I/O address lines decoded for 16-bit PC Card I/O windows

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	Address space (reset = memory, set = I/O)
Bit 1	Memory type (set = attribute) 16-bit PC Card memory windows only
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit) 16-bit PC Card windows only
Bit 4	Paged (set = true) 16-bit PC Card memory windows only
Bit 5	Shared (set = true) Not valid for CardBus PC Card memory windows
Bit 6	First Shared (set = true) Not valid for CardBus PC Card memory windows
Bit 7	Binding Specific Memory windows only
Bit 8	Card offsets are window sized (set = true) 16-bit PC Card memory windows only
Bit 9	Data path width (set = 32 bit / reset = see Bit 3) CardBus PC Card only
Bit 10	RESERVED (Reset to zero)
Bit 11 ··12	Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use.
Bits 13 ·· 15	Base Address Register number (1-7). CardBus PC Card only

The *AddressSpace* field describes the window type. When reset, a memory address space is requested. When set, an I/O address space is requested. The ability to specify I/O windows was added in this release of the Card Services specification. Prior releases of the specification required the **RequestIO** function be used to establish I/O windows. The **RequestIO** function is still supported for I/O address ranges within the first 64 KBytes of system I/O address space to promote backward compatibility. However, the preferred method is to use this service because it allows more flexible I/O window creation. For example, I/O address ranges may be specified anywhere within a 4 GByte address range, Base Address Registers on a CardBus PC Card may be specified, and multiple I/O ranges may be specified by calling this function more than once.

CardBus PC Cards do not have attribute memory so the *Memory Type* bit field shall never be set for CardBus PC Cards.

The *Enabled* bit indicates whether the window hardware should be enabled. A client may allocate multiple windows into the same system space as long as the client time-multiplexes them. Only one PC Card may respond to accesses into the shared system space at a time or hardware damage may result. (Note that CardBus PC Cards do not allow windows to share system space.)

If the *Data Path Width* is either 8 or 16 bits then field 9 is reset. If, however, field 9 is set, this overrides field 3 which can be ignored. Field 3 states whether the data path width is 8 or 16 bits and is never used by CardBus PC Cards. If field 9 is set this is a CardBus PC Card with a 32-bit interface and field 3 is ignored.

If the *Paged* bit is set to one, the window size must be a multiple of 16 KBytes. The first 16 KBytes of system memory address space used to map PC Card memory into system memory is referred to as page zero (0). The next 16 KBytes is page one (1), and so on. A 48 KByte window has three (3) pages numbered 0, 1, and 2. If the *Paged* bit is reset to zero, the window size is determined by the Size field. In both cases, the PC Card memory offset for the window is set by **MapMemPage**.

Normally, each request dedicates the requested system address range to the indicated socket, if it is available. However, for some applications and/or PC Cards, system address space may be shared by cards in two or more sockets. If the *Shared* bit is set, the range requested may be shared with another socket. *First Shared* is additionally set when a previously unshared system range is required that is intended to be shared with subsequent clients using this same system range. If a previously unshared system range is unavailable, the service fails and OUT_OF_RESOURCE is returned. The *Shared* bit may be used to allocate multiple windows mapping into the same system address space. This prevents Card Services from failing the request because the system address space has already been allocated to another window that was requested with *Shared* set to one. PC Cards must ensure that multiple cards mapped to the same system address don't all respond to accesses.

Card offsets are window sized is set by Card Services when the client must specify card offsets that are a multiple of the window size.

Prefetchable / Cacheable applies to CardBus PC Cards only. Clients accessing 16-bit PC Cards shall use zero (0) for this field.

The *Base Address Register* number indicates the associated Base Address Register on a CardBus PC Card. Base Address Register number seven (7) always refers to the Expansion ROM Base Address Register.

The *Base* field points to the physical location in system address space to map PC Card address space. If reset to zero (0) on entry, Card Services attempts to locate an available area of system address space. If successful, Card Services returns the base system address in this field. The *Size* field is the byte size of the window requested. *Size* may be zero to indicate that Card Services should provide the smallest size available.

Size and *Base* are in bytes, but must be of a supported granularity and alignment. If a client intends to map multiple windows into the same system address space, the first request should allow Card Services to determine the window base. Subsequent requests for windows using the same system address space MUST specify the base address returned by the first request. All windows should use the same size value. Only one request may set the *Enabled* bit in the *Attribute* field.

The *Access Speed* field, which is only used for 16-bit PC Card memory windows, is bit-mapped as follows:

Bits 0 ·· 2	Device speed code, if speed mantissa is zero Speed exponent, if speed mantissa is not zero
Bits 3 ·· 6	Speed mantissa
Bit 7	Wait (set = use **WAIT#**, if available)

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero (noted as reserved in the PCMCIA 2.1 / JEIDA 4.2 Release), the lower bits are a binary code representing a speed from the table below:

Code	Speed
0	(Reserved - do not use)
1	250 nsec
2	200 nsec
3	150 nsec
4	100 nsec
5 ·· 7	(Reserved - do not use)

The *IOAddrLines* field is the number of I/O address lines decoded by the PC Card in the specified socket. It is used by Card Services to determine whether any addresses outside the ranges specified needed to be marked as in-use because the combination of socket hardware and lines decoded could result in PC Card accesses outside the specified ranges. The *IOAddrLines* field is only used for 16-bit PC Card I/O windows.

The **MapMemPage** and **ModifyWindow** services use the values returned by this service for some types of windows.

SUCCESS is returned if a window has been defined.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to thirteen (13)
BAD_ATTRIBUTE	Specified attributes are invalid.
BAD_BASE	System memory address invalid
BAD_HANDLE	*ClientHandle* is invalid
BAD_SIZE	Window size is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
BAD_SPEED	Speed not supported
NO_CARD	No PC Card in socket
OUT_OF_RESOURCE	Internal data space is exhausted
IN_USE	Window requested is in use

5.54 ResetFunction (11H)

CardServices(ResetFunction, *ClientHandle, null, ArgLength, ArgPointer*)

This service resets the function of the PC Card in the specified socket. The *ClientHandle* returned by **RegisterClient** is passed in the *Handle* argument. The actual reset processing is done in the background asynchronously from the execution of this service. The client receives a **RESET_COMPLETE** upon completion of this processing.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bits 0 ·· 15	RESERVED (Reset to zero).

Card Services returns to the client after noting the request. As soon as the Card Services interface is available, Card Services sends **RESET_REQUEST** events. If any client rejects the **RESET_REQUEST**, Card Services terminates reset processing. Card Services then notifies the requesting client via a **RESET_COMPLETE** event with the Info argument set to the return code set by (any one of) the client(s) that rejected the request.

If no client rejects the reset request, Card Services sends a **RESET_PHYSICAL** to all clients that have indicated their interest in Reset events so they may prepare for a reset. When all clients have been notified, Card Services performs a reset. Card Services observes the appropriate wait after reset timing including monitoring the READY signal from the PC card and Card Services sends a **CARD_READY** notification if Card Services observes a transition to the READY state. Card Services sends a **CARD_RESET** notification to all registered clients.

Finally, Card Services notifies the requesting client's callback handler of a **RESET_COMPLETE** event. When control is returned from the background thread Card Services has been performing, the requesting client may continue processing.

When executing function reset events, Card Services must return the function to the same configuration state it had before the reset. If the function has been configured, then the configuration must be restored. It is also acceptable that Card Services implementations only perform resets on functions whose clients have released the configuration.

Those Card Services implementations which allow resets on configured functions must return the previously assigned resources to the client after the reset is successful. After performing a physical reset on a configured function, Card Services then returns the function to the last configuration obtained via a **RequestConfiguration** call. This will return to the client, for example, the memory, I/O, and power resources it previously requested. It is the responsibility of the client to perform any further configuration as required.

It is possible that a Card Services implementation may not allow resets to be performed on configured functions. In this case, Card Services must reject such **ResetFunction** requests with an **IN_USE** return

code. The client may then perform a **ReleaseConfiguration** and then re-issue the **ResetFunction** request. The function will then be in memory-only mode after the **ResetFunction** is complete. All of the resources, however, which the client has previously requested (e.g. memory windows, I/O windows, IRQs) will still be reserved for the client and if the client should now re-issue a **RequestConfiguration**, their original configuration will be returned to them.

The state of the function and socket during the time between the **RESET_PHYSICAL** and **CARD_RESET** events is undefined and may not be relied upon by clients. The client may rely upon the fact; however, that as part of the reset processing the function's configuration registers shall be reset.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_HANDLE	*ClientHandle* does not match owning client
BAD_SOCKET	*Socket* or function is invalid (socket/function request only)
IN_USE	This Card Services implementation does not permit configured cards to be reset. The configuration must first be released before resetting the card
NO_CARD	No PC Card in socket

5.55 ReturnSSEntry (23H)

CardServices(ReturnSSEntry, *null, null/SSEntry, ArgLength, ArgPointer*)

This service returns a pointer to an entry point that can be used to call Socket Services. The entry point is returned in the Pointer argument. The entry point references code in Card Services that calls the correct Socket Services entry point as required based on the physical adapter and socket numbers provided by the Socket Services client.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Information about the SS entry point

The *Attributes* field specifies details about the Socket Services entry point. It is defined the same as the *Attributes* field in **Add/ReplaceSocketServices**.

WARNING:

Directly accessing services provided by Socket Services which modify hardware state may cause the host system to crash. Clients should limit their access to services which only return state information.

Note: Making a request of Socket Services through this service is considered the same as a request through the Card Services interface. Subsequent Socket and Card Services requests are blocked until the prior request is complete. Card Services monitors requests made through this entry point to avoid performing asynchronous callback notifications when the interfaces are not available.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to two (2)
UNSUPPORTED_SERVICE	Implementation does not support service
UNSUPPORTED_MODE	Processor mode not supported

5.56 SetEventMask (31H)

CardServices(SetEventMask, *ClientHandle, null, ArgLength, ArgPointer*)

This service changes the event mask for the client. The *ClientHandle* returned by **RegisterClient** or **GetFirst/NextClient** is passed in the *Handle* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined below)
2	EventMask	2	I	N	Bit-mapped (defined below)
4	Socket	2	I	N	Logical socket

The *Attributes* field is bit-mapped. It identifies the type of event mask to be changed. The field is defined as follows:

Bit 0	Event mask of this socket only (set = true)
Bits 1 ·· 15	RESERVED (Reset to zero)

If Bit 0 is reset, the global event mask is changed. If Bit 0 is set, the event mask for this socket is changed. **RequestSocketMask** must have been requested by this client before the event mask for the socket can be set. BAD_HANDLE is returned if the client has not specifically registered for this socket.

The *Event Mask* field is bit-mapped. Card Services performs event notification based on this field. The low-order eight bits specify events noted by Socket Services. The upper eight bits specify events generated by Card Services. The field is defined as follows:

Bit 0	Write Protect
Bit 1	Card Lock Change
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Dead
Bit 5	Battery Low
Bit 6	Ready Change
Bit 7	Card Detect Change
Bit 8	PM Change
Bit 9	Reset
Bit 10	SS Update
Bit 11	Extended Status Change
Bits 12 ·· 15	RESERVED (reset to zero)

(See also ***3.4 Callback Interfaces***.)

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to six (6)
BAD_HANDLE	*ClientHandle* is invalid
BAD_SOCKET	*Socket* or function is invalid (socket/function request only) or this socket has not been requested via **RequestSocketMask**
NO_CARD	No PC Card in socket (socket requests, only)

5.57 SetRegion (29H)

CardServices(SetRegion, *null, null, ArgLength, ArgPointer*)

This service allows a client to set a PC Card region's characteristics. It is intended to allow region characteristics to be set when they are not available in the Card Information Structure.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Card Offset	4	I	N	Card Memory Region Offset
8	Region Size	4	I	N	Region Size
12	EffBlockSize	4	I	N	Erase Block Size
16	PartMultiple	2	I	N	Partition Multiple (Erase Block units)
18	JEDEC ID	2	I	N	Partition JEDEC Memory ID Code
20	Bias Offset	4	I	N	Address Bias for MTD
24	Access Speed	1	I	N	Window Speed Field

The *Socket* field identifies the logical socket containing the PC Card which has an undefined or incorrectly defined region. For 16-bit PC Cards, this field contains only the socket number (since Multiple Function 16-bit PC Cards provide multiple I/O functions combined with two standard memory spaces: Attribute and Common memory space). For CardBus PC Cards, the *Socket* identifies both the logical socket and function. The least significant byte is the logical socket. The most significant byte of the *Socket* field is the function. Allowable functions are numbered from 0 to 7.

The *Attributes* field is bit-mapped. The following bits are defined:

Bit 0	Memory type (set = attribute)
Bit 1	Delete Region (set = true)
Bit 2	RESERVED (Reset to zero)
Bit 3 · 4	Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use.
Bits 5 ·· 7	RESERVED (Reset to zero)
Bit 8	Virtual Region (set = true)
Bits 9 ·· 12	RESERVED (Reset to zero)
Bits 13 ·· 15	Base Address Register number (1-7).

CardBus PC Cards do not have attribute memory so that the *Memory Type* bit field must never be set for CardBus PC Cards.

Delete Region is set to one when Card Services should remove the last region for this PC Card's memory type from its internal region table. If the region specified is not the last region for this PC Card's memory type, BAD_OFFSET is returned. If an MTD is currently registered for this region, BAD_OFFSET is returned.

Note: If Delete Region is set to one, all fields after Region Size (offset 8) are ignored.

Virtual Region is set to one when the region can only be accessed via an appropriate MTD, i.e. the region is not addressable simply by presenting addresses to the PC Card (e.g. via a memory window).

Prefetchable / Cacheable applies to CardBus PC Cards only. 16-bit PC Cards shall use zero (0) for this field.

The *Base Address Register* number indicates the associated Base Address Register on the CardBus PC Card. Base Address Register number seven (7) always refers to the Expansion ROM Base Address Register.

The *Card Offset* through *JEDEC ID* fields are the same as defined in **GetFirstPartition**.

Once all of the regions on a PC Card requiring modification are successfully updated, the client should issue a **ResetFunction** request to notify registered MTDs they should re-evaluate whether they wish to handle any region on the card.

The *Bias Offset* field is used by Card Services to compute the address to pass to the supporting MTD. When a read, write, copy or erase request is made by a client, Card Services subtracts this value from the (relative) value passed by the client and adds the base offset passed to **OpenMemory** before giving the request to the MTD. This field should normally be zero for regions that are not virtual.

The value in the *Card Offset* field is used by Card Services during **OpenMemory** requests to determine the MTD supporting access to the memory.

A new region may be added by using a region number equal to the current number of regions on the PC Card.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to twenty-five (25)
BAD_ATTRIBUTE	Virtual region cannot be set due to existing physical region
BAD_OFFSET	Region offset is invalid
BAD_SIZE	Region size is invalid
BAD_SOCKET	*Socket* is invalid
BAD_SPEED	Speed is not supported
NO_CARD	No PC Card in socket

5.58 ValidateCIS (2BH)

CardServices(ValidateCIS, *null, null, ArgLength, ArgPointer*)

This service validates the Card Information Structure on the PC Card in the specified socket.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Chains	2	O	N	Number of chains validated

The *Socket* field identifies the logical socket and the function on the PC Card. The least significant byte is the logical socket. The most significant byte is the function. Single function PC Cards use a zero (0) value for the function. Multiple function PC Cards use a value between zero (0) and one less than the number of functions on the PC Card.

The *Chains* field returns the number of valid tuple chains located in the CIS. If zero (0) is returned, the CIS is not valid.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to four (4)
BAD_SOCKET	*Socket* is invalid
NO_CARD	No PC Card in socket

5.59 VendorSpecific (34H)

CardServices(VendorSpecific, *null, null, ArgLength, ArgPointer*)

This service is used to make vendor specific Card Services requests.

Offset	Field	Size	Type	Value	Detail/Description
0	InfoLen	2	I/O	N	Length of returned information in packet
2	Vendor Data	N	I/O	N	Vendor Specific Data

The *InfoLen* specifies the length of the valid portion of the *Vendor Data* passed and returned.

The *Vendor Data* is a vendor specific structure.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is less than two (2)
«Vendor specific»	«Vendor specifies valid return codes»

5.60 WriteMemory (24H)

CardServices(WriteMemory, *MemoryHandle, buffer, ArgLength, ArgPointer*)

This service writes data to a PC Card via the specified *MemoryHandle*. The *MemoryHandle* returned by **OpenMemory** is passed in the *Handle* argument. The pointer to the data buffer that contains the data to be written to the PC Card is passed in the *Pointer* argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Card Offset	4	I	N	Card Destination Address
4	Count	4	I	N	Number of bytes to transfer
8	Attributes	2	I	N	Bit-Mapped

The *Card Offset* is a relative offset from the physical offset provided to the **OpenMemory** request used to obtain the *MemoryHandle*. It is the location on the PC Card where the data should be written.

The *Count* field is the number of bytes to write to the PC Card.

The *Attributes* field is bit-mapped. The following bits are defined:

Bits 0 ·· 1	RESERVED (reset to zero)
Bit 2	Disable Erase (set to one = true)
Bit 3	Verify
Bits 4 ·· 15	RESERVED (reset to zero)

Disable Erase is set to one to request that the memory area not be erased before data is written to the PC Card. This erase is only done for requests that are erase block aligned and a multiple of erase blocks. *Verify* is set to one to request that the data written be verified after the write. If an MTD doesn't support verification, Card Services provides this support. **GetFirst/NextPartition/Region** can be used to determine the erase and verify capabilities of a memory area.

If the *MemoryHandle* identifies attribute memory, the client must access the attribute memory in an appropriate way. No special processing is done, i.e. all bytes requested are transferred, not just even bytes. This will typically mean that only single bytes on even address can be successfully written.

When used in a processor mode that supports segmentation (e.g. x86 architecture systems in 286 protected mode operation), all bytes transferred are required to be contained within the segment referenced by the pointer argument. This limits the count requested to be less than or equal to the maximum segment size.

Return Codes

BAD_ARG_LENGTH	*ArgLength* is not equal to ten (10)
BAD_HANDLE	Invalid memory area handle
BAD_OFFSET	Invalid destination offset
BAD_SIZE	Size of area to write is invalid
NO_CARD	No PC Card in socket
WRITE_FAILURE	Unable to write media
WRITE_PROTECTED	Media is write-protected

APPENDIX-A

6. SERVICE CODES

Table 6–1 Service Codes (by service)

Resource Management Services	
GetConfigurationInfo	04H
GetFirstWindow	37H
GetMemPage	39H
GetNextWindow	38H
MapMemPage	14H
ModifyConfiguration	27H
ModifyWindow	17H
ReleaseConfiguration	1EH
Reserved[1]	3BH
ReleaseIO	1BH
ReleaseIRQ	1CH
ReleaseSocketMask	2FH
ReleaseWindow	1DH
RequestConfiguration	30H
Reserved[1]	3AH
RequestIO	1FH
RequestIRQ	20H
RequestSocketMask	22H
RequestWindow	21H

Client Services	
DeregisterClient	02H
GetCardServicesInfo	0BH
GetEventMask	2EH
GetStatus	0CH
RegisterClient	10H
ResetFunction	11H
SetEventMask	31H

Client Utilities	
GetFirstPartition	05H
GetFirstRegion	06H
GetFirstTuple	07H
GetNextPartition	08H
GetNextRegion	09H
GetNextTuple	0AH
GetTupleData	0DH

Bulk Memory Services	
CheckEraseQueue	26H
CloseMemory	00H
CopyMemory	01H
DeregisterEraseQueue	25H
OpenMemory	18H
ReadMemory	19H
RegisterEraseQueue	0FH
WriteMemory	24H

Advanced Client Services	
AccessConfigurationRegister	36H
AddSocketServices	32H
AdjustResourceInfo	35H
GetClientInfo	03H
GetFirstClient	0EH
GetNextClient	2AH
MapLogSocket	12H
MapLogWindow	13H
MapPhySocket	15H
MapPhyWindow	16H
RegisterMTD	1AH
RegisterTimer	28H
ReleaseExclusive	2DH
ReplaceSocketServices	33H
RequestExclusive	2CH
ReturnSSEntry	23H
SetRegion	29H
ValidateCIS	2BH
VendorSpecific	34H

1. Do not use. Legacy feature no longer supported.

Table 6–2 Service Codes (sorted alphabetically)

Service	Code
AccessConfigurationRegister	36H
AddSocketServices	32H
AdjustResourceInfo	35H
CheckEraseQueue	26H
CloseMemory	00H
ConfigureFunction	3CH
CopyMemory	01H
DeregisterClient	02H
DeregisterEraseQueue	25H
GetCardServicesInfo	0BH
GetClientInfo	03H
GetConfigurationInfo	04H
GetEventMask	2EH
GetFirstClient	0EH
GetFirstPartition	05H
GetFirstRegion	06H
GetFirstTuple	07H
GetFirstWindow	37H
GetMemPage	39H
GetNextClient	2AH
GetNextPartition	08H
GetNextRegion	09H
GetNextTuple	0AH
GetNextWindow	38H
GetStatus	0CH
GetTupleData	0DH
InquireConfiguration	3DH
MapLogSocket	12H
MapLogWindow	13H
MapMemPage	14H
MapPhySocket	15H

Service	Code
MapPhyWindow	16H
ModifyConfiguration	27H
ModifyWindow	17H
OpenMemory	18H
ReadMemory	19H
RegisterClient	10H
RegisterEraseQueue	0FH
RegisterMTD	1AH
RegisterTimer	28H
ReleaseConfiguration	1EH
Reserved[1]	3BH
ReleaseExclusive	2DH
ReleaseIO	1BH
ReleaseIRQ	1CH
ReleaseSocketMask	2FH
ReleaseWindow	1DH
ReplaceSocketServices	33H
RequestConfiguration	30H
Reserved[1]	3AH
RequestExclusive	2CH
RequestIO	1FH
RequestIRQ	20H
RequestSocketMask	22H
RequestWindow	21H
ResetFunction	11H
ReturnSSEntry	23H
SetEventMask	31H
SetRegion	29H
ValidateCIS	2BH
VendorSpecific	34H
WriteMemory	24H

1. Do not use. Legacy feature no longer supported.

Table 6–3 Service Codes (sorted numerically)

Service	Code
CloseMemory	00H
CopyMemory	01H
DeregisterClient	02H
GetClientInfo	03H
GetConfigurationInfo	04H
GetFirstPartition	05H
GetFirstRegion	06H
GetFirstTuple	07H
GetNextPartition	08H
GetNextRegion	09H
GetNextTuple	0AH
GetCardServicesInfo	0BH
GetStatus	0CH
GetTupleData	0DH
GetFirstClient	0EH
RegisterEraseQueue	0FH
RegisterClient	10H
ResetFunction	11H
MapLogSocket	12H
MapLogWindow	13H
MapMemPage	14H
MapPhySocket	15H
MapPhyWindow	16H
ModifyWindow	17H
OpenMemory	18H
ReadMemory	19H
RegisterMTD	1AH
ReleaseIO	1BH
ReleaseIRQ	1CH
ReleaseWindow	1DH
ReleaseConfiguration	1EH

Service	Code
RequestIO	1FH
RequestIRQ	20H
RequestWindow	21H
RequestSocketMask	22H
ReturnSSEntry	23H
WriteMemory	24H
DeregisterEraseQueue	25H
CheckEraseQueue	26H
ModifyConfiguration	27H
RegisterTimer	28H
SetRegion	29H
GetNextClient	2AH
ValidateCIS	2BH
RequestExclusive	2CH
ReleaseExclusive	2DH
GetEventMask	2EH
ReleaseSocketMask	2FH
RequestConfiguration	30H
SetEventMask	31H
AddSocketServices	32H
ReplaceSocketServices	33H
VendorSpecific	34H
AdjustResourceInfo	35H
AccessConfigurationRegister	36H
GetFirstWindow	37H
GetNextWindow	38H
GetMemPage	39H
Reserved[1]	3AH
Reserved[1]	3BH
ConfigureFunction	3CH
InquireConfiguration	3DH

1. Do not use. Legacy feature no longer supported.

APPENDIX-B

7. EVENT CODES

Table 7–1 Event Codes (sorted alphabetically)

Event	Code	Source	Client(s)	Registered By
BATTERY_DEAD	01H	Hardware	Socket	RequestSocketMask
BATTERY_LOW	02H	Hardware	Socket	RequestSocketMask
CARD_INSERTION	40H	Hardware	All	RegisterClient
CARD_INSERTION [A]	40H	DeregisterMTD	MTDs	RegisterClient
CARD_INSERTION [A]	40H	RegisterClient	Requester	RegisterClient
CARD_INSERTION [A]	40H	ReleaseExclusive	All	RegisterClient
CARD_INSERTION [A]	40H	RequestExclusive	Requester	RequestExclusive
CARD_INSERTION [A]	40H	RequestExclusive	All	RegisterClient
CARD_LOCK	03H	Hardware	Socket	RequestSocketMask
CARD_READY	04H	Hardware	Socket	RequestSocketMask
CARD_REMOVAL	05H	Hardware	Socket	RequestSocketMask
CARD_REMOVAL [A]	05H	ReleaseExclusive	Socket	RequestSocketMask
CARD_REMOVAL [A]	05H	RequestExclusive	All	RegisterClient
CARD_RESET	11H	ResetFunction	Socket	RequestSocketMask
CARD_UNLOCK	06H	Hardware	Socket	RequestSocketMask
CLIENT_INFO	14H	GetClientInfo	Provider	RegisterClient
EJECTION_COMPLETE	07H	Hardware	Socket	RequestSocketMask
EJECTION_REQUEST	08H	Hardware	Socket	RequestSocketMask
ERASE_COMPLETE	81H	Queued Erase	Requester	RegisterEraseQueue
EXCLUSIVE_COMPLETE	0DH	RequestExclusive	Requester	RequestExclusive
EXCLUSIVE_REQUEST	0EH	RequestExclusive	Socket	RequestSocketMask
INSERTION_COMPLETE	09H	Hardware	Socket	RequestSocketMask
INSERTION_REQUEST	0AH	Hardware	Socket	RequestSocketMask
MTD_REQUEST	12H	Card Services	MTD	RegisterClient
PM_RESUME	0BH	Card Services	Socket	RequestSocketMask
PM_SUSPEND	0CH	Card Services	Socket	RequestSocketMask
REGISTRATION_COMPLETE	82H	RegisterClient	Requester	RegisterClient
REQUEST_ATTENTION	18H	Hardware	All	RegisterClient
RESET_COMPLETE	80H	ResetFunction	Requester	ResetFunction
RESET_PHYSICAL	0FH	ResetFunction	Socket	RegisterClient
RESET_REQUEST	10H	ResetFunction	Socket	RegisterClient
SS_UPDATED	16H	Card Services	All	RegisterClient
TIMER_EXPIRED	15H	Hardware	Requester	RegisterTimer
WRITE_PROTECT	17H	Hardware	All	RegisterClient

See the description of the table columns following the next page.

Table 7–2 Event Codes (sorted numerically)

Event	Code	Source	Client(s)	Registered By
BATTERY_DEAD	01H	Hardware	Socket	RequestSocketMask
BATTERY_LOW	02H	Hardware	Socket	RequestSocketMask
CARD_LOCK	03H	Hardware	Socket	RequestSocketMask
CARD_READY	04H	Hardware	Socket	RequestSocketMask
CARD_REMOVAL	05H	Hardware	Socket	RequestSocketMask
CARD_REMOVAL [A]	05H	ReleaseExclusive	Socket	RequestSocketMask
CARD_REMOVAL [A]	05H	RequestExclusive	All	RegisterClient
CARD_UNLOCK	06H	Hardware	Socket	RequestSocketMask
EJECTION_COMPLETE	07H	Hardware	Socket	RequestSocketMask
EJECTION_REQUEST	08H	Hardware	Socket	RequestSocketMask
INSERTION_COMPLETE	09H	Hardware	Socket	RequestSocketMask
INSERTION_REQUEST	0AH	Hardware	Socket	RequestSocketMask
PM_RESUME	0BH	Card Services	Socket	RequestSocketMask
PM_SUSPEND	0CH	Card Services	Socket	RequestSocketMask
EXCLUSIVE_COMPLETE	0DH	RequestExclusive	Requester	RequestExclusive
EXCLUSIVE_REQUEST	0EH	RequestExclusive	Socket	RequestSocketMask
RESET_PHYSICAL	0FH	ResetFunction	Socket	RegisterClient
RESET_REQUEST	10H	ResetFunction	Socket	RegisterClient
CARD_RESET	11H	ResetFunction	Socket	RequestSocketMask
MTD_REQUEST	12H	Card Services	MTD	RegisterClient
CLIENT_INFO	14H	GetClientInfo	Provider	RegisterClient
TIMER_EXPIRED	15H	Hardware	Requester	RegisterTimer
SS_UPDATED	16H	Card Services	All	RegisterClient
WRITE_PROTECT	17H	Hardware	All	RegisterClient
REQUEST_ATTENTION	18H	Hardware	All	RegisterClient
CARD_INSERTION [A]	40H	DeregisterMTD	MTDs	RegisterClient
CARD_INSERTION	40H	Hardware	All	RegisterClient
CARD_INSERTION [A]	40H	RegisterClient	Requester	RegisterClient
CARD_INSERTION [A]	40H	ReleaseExclusive	All	RegisterClient
CARD_INSERTION [A]	40H	RequestExclusive	Requester	RequestExclusive
CARD_INSERTION [A]	40H	RequestExclusive	All	RegisterClient
RESET_COMPLETE	80H	ResetFunction	Requester	ResetFunction
ERASE_COMPLETE	81H	Queued Erase	Requester	RegisterEraseQueue
REGISTRATION_COMPLETE	82H	RegisterClient	Requester	RegisterClient

See the description of the table columns on the next page.

Event Code Table column descriptions

Event [A] indicates an artificial event.

Code refers to the value present in the *Status* argument when the client's callback handler is invoked.

Source is what causes the event to occur:

- Hardware means a hardware causes the event.
- Card Services means Card Services generates the event after performing a request.
- Card Services service means using that service causes the event to be generated.

Client(s) refers to who is notified that the event has occurred:

- All means that all clients who have registered with Card Services using the **RegisterClient** service will receive notification of the event.
- MTDs means that all clients who have registered with Card Services using the **RegisterClient** service with the MTD bit set, will receive notification of the event.
- Requester means that clients who have requested an erase background service will receive notification of completion at the address they specified in the erase queue header. Clients that requested a **RegisterClient** will receive notification of completion at the address specified in the request packet.
- Provider means that the client providing the client information is notified of the event.
- Socket means that clients who have registered with Card Services to use a PC Card in a specific socket with the **RequestSocketMask** service will receive notification of the event when it occurs on the specified socket. Clients that have used **RegisterClient** and enabled the socket events will also be notified.

Registered By identifies the Card Services service that registers a client to receive the event notification. Notification for most events can be enabled/disabled on a client and socket basis. This column indicates the expected usage, however, **SetEventMask** can be used to change whether a client gets notified for all sockets or just particular sockets.

APPENDIX-C

8. RETURN CODES

Table 8–1 Return Codes (sorted alphabetically)

Return Code	Value	Description
«Reserved»	05H, 0CH, 10H, 13H	«Reserved for historical purposes»
BAD_ADAPTER	01H	Specified adapter is invalid
BAD_ARG_LENGTH	1BH	*ArgLength* argument is invalid
BAD_ARGS	1CH	Values in Argument Packet are invalid
BAD_ATTRIBUTE	02H	Value specified for attributes field is invalid
BAD_BASE	03H	Specified base system memory address is invalid
BAD_EDC	04H	Specified EDC generator is invalid
BAD_HANDLE	21H	ClientHandle is invalid
BAD_IRQ	06H	Specified IRQ level is invalid
BAD_OFFSET	07H	Specified PC Card memory array offset is invalid
BAD_PAGE	08H	Specified page is invalid
BAD_SIZE	0AH	Specified size is invalid
BAD_SOCKET	0BH	Specified socket is invalid (logical or physical)
BAD_SPEED	17H	Specified speed is unavailable
BAD_TYPE	0DH	Window or interface type specified is invalid
BAD_VCC	0EH	Specified **VCC** power level index is invalid
BAD_VERSION	22H	Client version is unsupported
BAD_VPP	0FH	Specified **VPP1** or **VPP2** power level index is invalid
BAD_WINDOW	11H	Specified window is invalid
BUSY	18H	Unable to process request at this time - retry later
CONFIGURATION_LOCKED	1DH	A configuration has already been locked
GENERAL_FAILURE	19H	An undefined error has occurred
IN_USE	1EH	Requested resource is being used by a client
NO_CARD	14H	No PC Card in socket
NO_MORE_ITEMS	1FH	There are no more of the requested item
OUT_OF_RESOURCE	20H	Card Services has exhausted resource
READ_FAILURE	09H	Unable to complete read request
SUCCESS	00H	The request succeeded.
UNSUPPORTED_MODE	16H	Processor mode is not supported
UNSUPPORTED_SERVICE	15H	Implementation does not support service
WRITE_FAILURE	12H	Unable to complete write request
WRITE_PROTECTED	1AH	Media is write-protected

Return Codes common to Socket Services use the same values. *Italicized* entries are reserved for historical purposes and should not be used. Return codes above 19H are unique to Card Services.

Table 8–2 Return Codes (sorted numerically)

Return Code	Value	Description
SUCCESS	00H	The request succeeded.
BAD_ADAPTER	01H	Specified adapter is invalid
BAD_ATTRIBUTE	02H	Value specified for attributes field is invalid
BAD_BASE	03H	Specified base system memory address is invalid
BAD_EDC	04H	Specified EDC generator is invalid
«Reserved»	05H	«Reserved for historical purposes»
BAD_IRQ	06H	Specified IRQ level is invalid
BAD_OFFSET	07H	Specified PC Card memory array offset is invalid
BAD_PAGE	08H	Specified page is invalid
READ_FAILURE	09H	Unable to complete read request
BAD_SIZE	0AH	Specified size is invalid
BAD_SOCKET	0BH	Specified socket is invalid (logical or physical)
«Reserved»	0CH	«Reserved for historical purposes»
BAD_TYPE	0DH	Window or interface type specified is invalid
BAD_VCC	0EH	Specified **VCC** power level index is invalid
BAD_VPP	0FH	Specified **VPP1** or **VPP2** power level index is invalid
«Reserved»	10H	«Reserved for historical purposes»
BAD_WINDOW	11H	Specified window is invalid
WRITE_FAILURE	12H	Unable to complete write request
«Reserved»	13H	«Reserved for historical purposes»
NO_CARD	14H	No PC Card in socket
UNSUPPORTED_SERVICE	15H	Implementation does not support service
UNSUPPORTED_MODE	16H	Processor mode is not supported
BAD_SPEED	17H	Specified speed is unavailable
BUSY	18H	Unable to process request at this time - retry later
GENERAL_FAILURE	19H	An undefined error has occurred
WRITE_PROTECTED	1AH	Media is write-protected
BAD_ARG_LENGTH	1BH	*ArgLength* argument is invalid
BAD_ARGS	1CH	Values in Argument Packet are invalid
CONFIGURATION_LOCKED	1DH	A configuration has already been locked
IN_USE	1EH	Requested resource is being used by a client
NO_MORE_ITEMS	1FH	There are no more of the requested item
OUT_OF_RESOURCE	20H	Card Services has exhausted resource
BAD_HANDLE	21H	ClientHandle is invalid
BAD_VERSION	22H	Client version is unsupported

Return Codes common to Socket Services use the same values. *Italicized* entries are reserved for historical purposes and should not be used. Return codes above 19H are unique to Card Services.

APPENDIX-D

9. BINDINGS

9.1 Overview

A Card Services binding answers the following five questions for a specific host environment:

How is the presence of Card Services determined?

How are Card Services requests made?

How are arguments passed to and from Card Services?

How are binding-specific arguments and services defined?

How are arguments passed to and from client callback handlers?

A specific host environment for a Card Services client is defined by the operating system in use and the host platform's architecture. Multi-mode processors may require separate bindings for each mode used by an operating system. Operating systems that emulate other operating systems may also implement more than one Card Services binding.

9.2 Presence Detection

A client determines whether Card Services is available in the host environment through a binding specific presence detection mechanism. All bindings specify a method of determining the presence of Card Services using operations that have well-defined responses whether Card Services is actually installed or not. A Card Services client may use the Card Services request mechanism for presence detection if the binding guarantees a negative response is returned if Card Services is not installed.

9.3 Making Card Services Requests

Card Services requests are made in a binding-specific manner. Software interrupts, far or near calls, operating system device driver interfaces and other methods of making requests of Card Services may be appropriate depending on the host environment. Environments which emulate other environments may actually provide more than one method of making a Card Services request. If a Card Services implementation is not able to satisfy a request from a client in an emulated environment, it must insure the request is failed.

9.4 Argument Passing

A Card Services binding defines how arguments are passed to and from Card Services. Depending on the host environment, arguments may be passed in registers, in stack-based packets or even in global data areas. There are six possible input arguments to a Card Services request.

Service	The service Card Services is being requested to perform.
Handle	An implementation specific value identifying a Card Services object (Client, Window, etc.).
Pointer	A binding specific pointer.
ArgLength	The length of the *ArgPacket*.
ArgPointer	A binding specific pointer to an *ArgPacket*.
ArgPacket	A request specific data packet.

Many services provided by the Card Services interface do not use all of the arguments. If an input argument is not used for a service, it is ignored.

Services provided by the Card Services interface may modify the *ArgPacket* to return information. Services may also use the *Handle* and *Pointer* arguments to return information.

If a services provided by the Card Services interface does not use an argument to return information, it is returned unmodified.

All services provided by the Card Services interface return *Status*. This is a **Return Code** as defined by Appendix C. A binding may use the same or overlapping representations for the **Service** input argument and *Status*.

9.5 Binding Specific Arguments and Services

The *Pointer* and *ArgPointer* arguments are binding specific. They may be far or near pointers or even an index, if global data areas are being used by the binding.

Four (4) services provided by the Card Services interface have different definitions depending on the client's environment. They are:

AddSocketServices (32H)	*Attributes*
RegisterClient (10H)	*Client Data*
ReplaceSocketServices (33H)	*Attributes*
RequestWindow (21H)	*Attributes*

Clients use the binding specific data areas as described in the sections that follow. Only the unshaded areas of tables are binding specific. Shaded areas are the same in all environments. Included text indicates binding specific changes from the original service definition. An ellipse (…) is used to indicate text from the original service definition is unchanged and is not repeated in the binding specific section.

9.6 Client Callback Handler

Card Services performs asynchronous event notifications to registered clients. A binding specifies how arguments are passed to and from a client callback handler. As with Card Services request arguments, client callback arguments may be passed in registers, in stack-based packets or even in global data areas. There are seven possible input arguments to a client callback handler:

Event	The type of event notification being made to the client.
Socket/Function	The socket and function experiencing the event.
Info	An event specific value.
MTDRequest	A binding specific pointer to an MTD request packet.
Buffer	A binding specific pointer to an event specific buffer.
Misc	An event specific value.
Client Data	Data provided previously to Card Services in the arguments of the client's **RegisterClient** request.

Many event notifications do not use all of the arguments. If an input argument is not used by Card Services for an event notification, it must be ignored by the client.

An event notification never uses both the **Buffer** and **Misc** arguments. A binding may use the same or overlapping representations for these arguments. Client callback handlers may modify the **Buffer** to return information from a CLIENT_INFO event notification.

If an argument is not used to return information, the client callback handler must preserve the argument and return it unmodified to Card Services. Some event notifications require *Status* to be returned by the client callback handler. This is a **Return Code** as defined by Appendix C.

9.7 x86 Architecture Bindings

This section describes the Card Services bindings for x86 architecture systems.

There are a number of members of the x86 processor family offering up to three modes of operations: real, protect and virtual (also known as V86). The x86 family also varies in addressable memory space (1, 16 or 4096 megabytes), register size (16 or 32-bit) and memory management capabilities (paging).

Processor	Register Size	Address Space	Real	Protect	Virtual	Paging
x86	16	1 MB	Yes	No	No	No
286	16	16 MB	Yes	Yes	No	No
386 and above	32	4096 MB	Yes	Yes	Yes	Yes

A real mode client is limited to one megabyte of address space and 16-bit registers. In protect mode, clients can address much larger amounts of memory with 16 or, on some processors, 32-bit registers.

In V86 mode, multiple real mode clients operate independently as if they were the only real mode client. A control program running in protect mode remaps memory space so each client believes it is operating in the first megabyte of address space, addressing physical memory.

Different operating systems exploit different features of these processors. Due to the differences between the capabilities of x86 processors and the manner that x86 operating systems use the processor, this section actually defines four separate bindings.

The different bindings are based on the type of client requiring Card Services support. An environment must provide a binding for each type of client it supports. The six types of clients defined by this section are:

DOS real mode clients
OS/2 16-bit protect mode clients
Windows NT 4.0 Kernel Mode clients
Windows 16-bit protect mode clients
Windows 32-bit protect mode VxD clients
Win32 DLL Clients

9.7.1 DOS Real Mode Clients

This binding is used by DOS real mode clients, DOS clients using 'extender' technology to operate in protect mode and DOS clients running in a virtual machine under OS/2 and Windows as a V86 task.

DOS real mode clients typically believe they are the only process in the host platform. They expect to address physical memory by using segment addresses in the segment registers. They also use 16-bit wide registers and assume there is no memory management or paging being performed by the processor.

Some DOS real mode clients use extenders to operate in protect-mode. Performing most of their activities in protect mode with larger address spaces and possibly larger registers, they shift down to real-mode or V86 mode to access DOS, ROM BIOS or DOS-based device drivers and Terminate and Stay Resident (TSR) programs. Even though these clients are actually operating in protect mode, their interaction with Card Services (and other DOS-based services) is as a real mode or V86 client using 16-bit registers and what they believe to be physical memory addressing.

Environments which emulate the DOS environment must translate requests from DOS clients into the appropriate format for the native environment of Card Services.

9.7.1.1 Presence Detection

DOS real mode clients detect the presence of Card Services by performing a Card Services **GetCardServicesInfo** request. If this request returns SUCCESS, the presence of Card Services is validated by confirming the *Signature* field of the *ArgPacket* contains the proper values. If the request returns with other than SUCCESS or the *Signature* field does not contain the proper values, the client must assume Card Services is not present.

9.7.1.2 Making Card Services Requests

DOS real mode clients make Card Services requests by placing the appropriate values in the registers indicated below, placing the value AFH in the [AH] register and performing an INT 1AH.

9.7.1.3 Argument Passing

DOS real mode clients pass all input arguments for Card Services requests in the following registers:

[AH]	**CS ID**	AFH (as described above)
[AL]	**Service**	See **Appendix A, Service Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit segment, [SI] is the 16-bit offset
[CX]	**ArgLength**	
[ES]:[BX]	**ArgPointer**	

Card Services returns the following registers:

[CF]	**Success/Fail**	If [CF] reset request succeeded, If [CF] set request failed
[AX]	**Status**	See **Appendix C, Return Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit segment, [SI] is the 16-bit offset
[CX]	**ArgLength**	(Unchanged from input)
[ES]:[BX]	**ArgPointer**	(Unchanged from input)

9.7.1.4 Binding Specific Arguments and Services

All *Pointer* and *ArgPointer* services arguments are far pointers. DOS real mode clients use segment:offset addresses to point to what they believe is physical memory. As noted above, DOS real mode clients operating in V86 mode may have their memory space re-mapped by a protect mode control program. A Card Services implementation is responsible for transparently performing any address translation that may be required.

AddSocketServices (32H)

CardServices(**AddSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services *real mode* entry point. Card Services *makes a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	ZERO	*Zero (0) indicating the SSEntry and DataPointer arguments are real mode segment:offset pointers.*
2	DataPointer	4	I	N	*Pointer to Socket Services data area. Segment:offset, stored in little-endian format (offset first)*

The *Attributes* field must be reset to zero (0) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are real mode segment:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to six (6).

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined elsewhere).
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in [DI] for event notification.
6	Client Data Segment	2	I	N	Segment of client's data area. Placed in [DS] before calling client callback handler for event notification.
8	Client Data Offset	2	I	N	Reference data returned to client in [SI] for event notification.
10	Reserved	2	I	ZERO	Reserved (must be reset to zero).
12	Version	2	I	BCD	the *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

CardServices(**ReplaceSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to replace an existing one that Card Services is already using. The new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The *Pointer* argument contains the Socket Services *real mode* entry point. Card Services *makes a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	ZERO	*Zero (0) indicating the SSEntry and DataPointer arguments are real mode segment:offset pointers.*
6	DataPointer	4	I	N	*Pointer to Socket Services data area. Segment:offset, stored in little-endian format (offset first).*

…

The *Attributes* field must be reset to zero (0) to indicate the Socket Services entry point in the Pointer argument and the DataPointer fields are real mode segment:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH — *ArgLength* not equal to ten (10).

…

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11 ·· 12	Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use.
Bits 13 ·· 15	Base Address Register number (1-7). CardBus PC Card only

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.1.5 Client Callback Handler

DOS real mode clients pass all event notification arguments for client callback handlers in the following registers:

[AH]	**Reserved**	Reserved (reset to zero)
[AL]	**Event**	See **Appendix B, Event Codes**
[BX]	**Misc**	An event specific value (when **Buffer** not used)
[ES]:[BX]	**Buffer**	Far pointer to event specific buffer (when **Misc** not used)
[CX]	**Socket/Function**	Socket and function experiencing the event, if applicable
[DX]	**Info**	An event specific value
[SI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DS]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)

The client callback handler returns the following registers:

[CF]	**Success/Fail**	If [CF] reset event notification succeeded, If [CF] set event notification failed
[AX]	**Status**	See **Appendix C, Return Codes**

9.7.2 OS/2 16-bit Protect Mode Clients

This binding is used by OS/2 16-bit protect mode clients.

9.7.2.1 Presence Detection

OS/2 16-bit protect mode clients detect the presence of Card Services by performing an OS/2 **AttachDD** DevHlp function with the device driver name "PCMCIA$ ". (PCMCIA$ followed by a single blank). If successful, this request returns the inter-device driver communication (IDC) entry point for Card Services. If unsuccessful, the client must assume that Card Services is not present.

9.7.2.2 Making Card Services Requests

OS/2 16-bit protect mode clients make Card Services requests by placing the appropriate values in the registers indicated below and making a FAR CALL to the entry point returned by the **AttachDD** request described for presence detection above.

9.7.2.3 Argument Passing

OS/2 16-bit protect mode clients pass all input arguments for Card Services requests in the following registers:

[AH]	**CS ID**	AFH (as described above)
[AL]	**Service**	See **Appendix A, Service Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit selector, [SI] is the 16-bit offset
[CX]	**ArgLength**	
[ES]:[BX]	**ArgPointer**	In selector:offset form

Card Services returns the following registers:

[CF]	**Success/Fail**	If [CF] reset request succeeded, If [CF] set request failed
[AX]	**Status**	See **Appendix C, Return Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit selector, [SI] is the 16-bit offset
[CX]	**ArgLength**	(Unchanged from input)
[ES]:[BX]	**ArgPointer**	(Unchanged from input)

9.7.2.4 Binding Specific Arguments and Services

All *Pointer* and *ArgPointer* services arguments are far pointers. OS/2 16-bit protect mode clients use selector:offset addresses. A Card Services implementation is responsible for transparently performing any address translation that may be required.

AddSocketServices (32H)

CardServices(**AddSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services *16-bit protect mode* entry point. Card Services *makes a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	1	*One (1) indicating the SSEntry and DataPointer arguments are far 16-bit selector:offset pointers.*
2	DataPointer	4	I	N	*Pointer to Socket Services data area. Selector:offset, stored in little-endian format (offset first).*

The *Attributes* field must be set to one (1) to indicate the Socket Services entry point in the Pointer argument and the DataPointer field are far 16-bit selector:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to six (6).

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined elsewhere).
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in [DI] for event notification.
6	Client Data Selector	2	I	N	Selector for client's data area. Placed in [DS] before calling client callback handler for event notification.
8	Client Data Offset	2	I	N	Reference data returned to client in [SI] for event notification.
10	Reserved	2	I	ZERO	Reserved (must be reset to zero).
12	Version	2	I	BCD	The *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

CardServices(**ReplaceSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to replace an existing one that Card Services is already using. The new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The *Pointer* argument contains the Socket Services *16-bit protect mode* entry point. Card Services makes *a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	1	*One (1) indicating the SSEntry and DataPointer arguments are far 16-bit selector:offset pointers.*
6	DataPointer	4	I	N	*Pointer to Socket Services data area. Selector:offset, stored in little-endian format (offset first).*

…

The *Attributes* field must be set to one (1) to indicate the Socket Services entry point in the Pointer argument and the DataPointer field are far 16-bit protect mode selector:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH *ArgLength* not equal to ten (10).

…

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11	Pre-fetch (set = prefetchable)
Bit 12	Cache (set = cacheable)
Bits 13 ·· 15	Base Address Register number (1-7). CardBus PC Card only

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.2.5 Client Callback Handler

OS/2 16-bit protect mode clients pass all event notification arguments for client callback handlers in the following registers:

[AH]	**Reserved**	Reserved (reset to zero)
[AL]	**Event**	See **Appendix B, Event Codes**
[BX]	**Misc**	An event specific value (when **Buffer** not used)
[ES]:[BX]	**Buffer**	Far 16-bit protect mode selector:offset pointer to an event specific buffer (when **Misc** not used)
[CX]	**Socket/Function**	Socket and function experiencing the event, if applicable
[DX]	**Info**	An event specific value
[SI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DS]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)

The client callback handler returns the following registers:

[CF]	**Success/Fail**	If [CF] reset event notification succeeded, If [CF] set event notification failed
[AX]	**Status**	See **Appendix C, Return Codes**

9.7.3 Windows 16-bit Protect Mode Clients

This binding is used by Windows 16-bit protect mode clients. It is an extension of the DOS real mode client binding to support protect mode addressing. It may also be used by DOS 16-bit protect mode clients running in a Windows virtual machine that do not switch to V86 mode to make DOS real mode compatible use of Card Services.

It should also be noted that Windows operates on two modes: Standard and Enhanced. This binding is intended to define the Card Services Windows 16-bit protect mode client interface for both Standard and Enhanced mode Windows.

Environments which emulate the Windows environment must translate requests from Windows 16-bit protect mode clients into the appropriate format for the native environment of Card Services.

9.7.3.1 Presence Detection

Windows 16-bit protect mode clients detect the presence of Card Services by performing a Card Services **GetCardServicesInfo** request. If this request returns SUCCESS, the presence of Card Services is validated by confirming the *Signature* field of the *ArgPacket* contains the proper values. If the request returns with other than SUCCESS or the *Signature* field does not contain the proper values, the client must assume Card Services is not present.

9.7.3.2 Making Card Services Requests

Windows 16-bit protect mode clients make Card Services requests by placing the appropriate values in the registers indicated below, placing the value AFH in the [AH] register and performing an INT 1AH.

9.7.3.3 Argument Passing

Windows 16-bit protect mode clients pass all input arguments for Card Services requests in the following registers:

[AH]	**CS ID**	AFH (as described above)
[AL]	**Service**	See **Appendix A, Service Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit selector, [SI] is the 16-bit offset
[CX]	**ArgLength**	
[ES]:[BX]	**ArgPointer**	In selector:offset format

Card Services returns the following registers:

[CF]	**Success/Fail**	If [CF] reset request succeeded, If [CF] set request failed
[AX]	**Status**	See **Appendix C, Return Codes**
[DX]	**Handle**	
[DI]:[SI]	**Pointer**	[DI] is the 16-bit segment, [SI] is the 16-bit offset
[CX]	**ArgLength**	(Unchanged from input)
[ES]:[BX]	**ArgPointer**	(Unchanged from input)

9.7.3.4 Binding Specific Arguments and Services

All *Pointer* and *ArgPointer* services arguments are far pointers. Windows 16-bit protect mode clients use selector:offset addresses. A Card Services implementation is responsible for transparently performing any address translation that may be required.

AddSocketServices (32H)

CardServices(**AddSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services *16-bit protect mode* entry point. Card Services *makes a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	1	*One (1) indicating the SSEntry and DataPointer arguments are far 16-bit protect mode pointers.*
2	DataPointer	4	I	N	*Pointer to Socket Services data area. Selector:offset, stored in little-endian format (offset first)*

The *Attributes* field must be set to one (1) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* field are far 16-bit protect mode selector:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to six (6).

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped.
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in [DI] for event notification.
6	Client Data Segment	2	I	N	Selector for client's data area. Placed in [DS] before calling client callback handler for event notification.
8	Client Data Offset	2	I	N	Reference data returned to client in [SI] for event notification.
10	Reserved	2	I	ZERO	Reserved (must be reset to zero).
12	Version	2	I	BCD	The *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

CardServices(**ReplaceSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to replace an existing one that Card Services is already using. The new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The *Pointer* argument contains the Socket Services *16-bit protect mode* entry point. Card Services *makes a FAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	1	*One (1) indicating the SSEntry and DataPointer arguments are far 16-bit protect mode pointers.*
6	DataPointer	4	I	N	*Pointer to Socket Services data area. Selector:offset, stored in little-endian format (offset first)*

…

The *Attributes* field must be reset to one (1) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are far 16-bit protect mode selector:offset pointers.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [DS] and [SI] registers when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH — *ArgLength* not equal to ten (10).

…

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11	Pre-fetch (set = prefetchable)
Bit 12	Cache (set = cacheable)
Bits 13 ·· 15	Base Address Register number (1-7). CardBus PC Card only

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.3.5 Client Callback Handler

DOS real mode clients pass all event notification arguments for client callback handlers in the following registers:

[AH]	**Reserved**	Reserved (reset to zero)
[AL]	**Event**	See **Appendix B, Event Codes**
[BX]	**Misc**	An event specific value (when **Buffer** not used)
[ES]:[BX]	**Buffer**	Far 16-bit protect mode selector:offset pointer to an event specific buffer (when **Misc** not used)
[CX]	**Socket/Function**	Socket and function experiencing the event, if applicable
[DX]	**Info**	An event specific value
[SI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DS]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)

The client callback handler returns the following registers:

[CF]	**Success/Fail**	If [CF] reset event notification succeeded, If [CF] set event notification failed
[AX]	**Status**	See **Appendix C, Return Codes**

9.7.4 Windows Flat 32-bit Protect Mode VxD Clients

This binding is used by Virtual Device Driver (VxD) clients supporting Windows Enhanced mode. Windows VxDs operate in 32-bit protect mode using a flat memory model. In the flat memory model, all of the x86 segments registers contain selectors describing the same four gigabyte address range.

A Windows VxD client must include the following service table:

```
Begin_Service_Table     PCCARD
PCCARD_Service          PCCARD_Get_Version
PCCARD_Service          PCCARD_DoRequest
End_Service_Table       PCCARD
```

9.7.4.1 Presence Detection

Windows VxD clients detect the presence of Card Services by using the VxD service interface and performing a Get_Version request. If the request returns with the [CF] set, the client must assume Card Services is not present. If the request returns with the [CF] reset, the version of Card Services present is returned in the [AX] register.

9.7.4.2 Making Card Services Requests

Windows VxD clients make Card Services requests using the VxD service interface through the VxDcall macro. Card Services requests are made by placing the required values in the registers described below and using the following instruction:

```
VxDcall         PCCARD_DoRequest
```

9.7.4.3 Argument Passing

Windows VxD clients pass all input arguments for Card Services requests in the following registers:

[AH]	**CS ID**	AFH (as described above)
[AL]	**Service**	See **Appendix A, Service Codes**
[DX]	**Handle**	
[ESI]	**Pointer**	32-bit flat offset
[CX]	**ArgLength**	
[EBX]	**ArgPointer**	32-bit flat offset

Card Services returns the following registers:

[CF]	**Success/Fail**	If [CF] reset request succeeded, If [CF] set request failed
[AX]	**Status**	See **Appendix C, Return Codes**
[DX]	**Handle**	
[ESI]	**Pointer**	32-bit flat offset
[CX]	**ArgLength**	(Unchanged from input)
[EBX]	**ArgPointer**	(Unchanged from input)

9.7.4.4 Binding Specific Arguments and Services

All *Pointer* and **ArgPointer** services arguments are 32-bit offsets.

AddSocketServices (32H)

CardServices(**AddSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services *32-bit flat protect mode* entry point. Card Services *makes a NEAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	3	*Three (3) indicating the SSEntry and DataPointer arguments are 32-bit flat protect mode offsets.*
2	DataPointer	4	I	N	*32-bit flat protect mode offset of the Socket Services data area, stored in little-endian format.*

The *Attributes* field must be set to three (3) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are 32-bit flat protect mode offsets.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [ESI] register when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH	*ArgLength* not equal to six (6).

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped.
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in [DI] for event notification.
6	Reserved	2	I	ZERO	Reserved (must be reset to zero).
8	Client Data Offset	4	I	N	32-bit flat protect mode offset of client's data area. Placed in [ESI] before calling client callback handler for event notification.
12	Version	2	I	BCD	The *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

`CardServices`(**`ReplaceSocketServices`**, *`null`*, *`SSEntry`*, *`ArgLength`*, *`ArgPointer`*)

This service allows a new Socket Services handler to replace an existing one that Card Services is already using. The new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The *Pointer* argument contains the Socket Services *32-bit flat protect mode* entry point. Card Services *makes a NEAR CALL to* Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	3	*Three (3) indicating the SSEntry and DataPointer arguments are 32-bit flat protect mode offsets.*
6	DataPointer	4	I	N	*32-bit flat protect mode offset of Socket Services data area, stored in little-endian format.*

…

The *Attributes* field must be set to three (3) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are 32-bit flat protect mode offsets.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [ESI] register when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH — *ArgLength* not equal to ten (10).

…

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11	Pre-fetch (set = prefetchable)
Bit 12	Cache (set = cacheable)
Bits 13 ·· 15	RESERVED (reset to zero)

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.4.5 Client Callback Handler

DOS real mode clients pass all event notification arguments for client callback handlers in the following registers:

[AH]	**Reserved**	Reserved (reset to zero)
[AL]	**Event**	See **Appendix B, Event Codes**
[BX]	**Misc**	An event specific value (when **Buffer** not used)
[EBX]	**Buffer**	32-bit flat protect mode offset of an event specific buffer (when **Misc** not used)
[CX]	**Socket/Function**	Socket and function experiencing the event, if applicable
[DX]	**Info**	An event specific value
[ESI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)
[DI]	**Client Data**	From **RegisterClient** *ArgPacket* (see above)

The client callback handler returns the following registers:

[CF]	**Success/Fail**	If [CF] reset event notification succeeded, If [CF] set event notification failed
[AX]	**Status**	See **Appendix C, Return Codes**

9.7.5 Win32 DLL Clients

This binding is used by Win32 DLL clients.

9.7.5.1 Presence Detection

Win32 DLL clients detect the presence of Card Services by attaching to the Card Services DLL and either using the OpenCS entry point to the DLL or by simply issuing a GetCardServicesInfo request via the DoCardServices entry point.

WARNING: ANY CLIENTS THAT UTILIZE THE OPENCARDSERVICES REQUEST MUST ALSO USE THE CLOSECARDSERVICES REQUEST WHEN COMPLETE!

9.7.5.2 Making Card Services Requests

Win32 DLL clients make Card Services requests by properly filling in the parameters for the argument packet and then invoking the DoCardServices entry point of the DLL.

9.7.5.3 Argument Passing

Win32 DLL clients pass all input arguments for Card Services requests using a C-style call that exactly matches the definition of the Card Services request descriptions:

```
enum RetCode _cdecl
             DoCardServices (enum CS_SUBFUNC, WORD & handle, FPTR & Far
             Pointer, uint ArgLength, FPTR ArgBuffer) ;
```

Win32 DLL MTD clients that need to make MTDHelper requests of Card Services do so using a C-style call:

```
enum RetCode cdecl DoMTD (MTD_REGS *);
```

9.7.5.4 Binding Specific Arguments and Services

All *Pointer* and **ArgPointer** services arguments are C-style Far Pointers (32-bit offsets).

AddSocketServices (32H)

…

The AddSocketServices request is not expected to be used by Win32 DLL clients. The only valid return code is UNSUPPORTED_FUNCTION.

Return Codes

UNSUPPORTED_FUNCTION This service is not available for Win32 DLL Clients.

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped.
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in *ClientData* for event notification.
6	Reserved	2	I	ZERO	Reserved (must be reset to zero).
8	Client Data Offset	4	I	N	32-bit offset of client's data area. Placed in *ClientOffset* before calling client callback handler for event notification.
12	Version	2	I	BCD	The *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

…

The ReplaceSocketServices request is not expected to be used by Win32 DLL clients. The only valid return code is UNSUPPORTED_FUNCTION.

Return Codes

UNSUPPORTED_FUNCTION	This service is not available for Win32 DLL Clients.

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11	Pre-fetch (set = prefetchable)
Bit 12	Cache (set = cacheable)
Bits 13 ·· 15	RESERVED (reset to zero)

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.5.5 Client Callback Handler

Win32 DLL clients pass all event notification arguments for client callback handlers in the following structure:

```
typedef struct tagCallBackRegs         // Call back register structure
{
      DWORD    dwReserved1;                  // Reserved, not used
      union
      {
         DWORD dwReserved2;                  // Reserved, not used
         WORD  wClientData;
      }
      DWORD dwClientOff;
      DWORD dwMTDRegOff;
      DWORD    dwReserved3;                  // Reserved, not used
      union
      {
         WORD  wMisc;                        // Misc
         void *pBuffer;                      // Buffer (EBX)
      };
      union
      {
         DWORD dwReserved4;                  // Reserved, not used
         WORD  wInfo;                        // Info
      };
      union
      {
         DWORD dwReserved5;                  // Reserved, not used
         union
         {
               WORD  hLogSkt;                // Logical socket
               struct
               {
                     BYTE bLogSkt;           // Logical socket number
                     BYTE bSktFunction;      // Logical function number
               }
         }
      };
      union
      {
         DWORD dwReserved6;                  // Reserved, not used
         WORD   wEvent;                      // CS Event
         enum RETCODE wStatus;               // CS Status
      };
} CB_REGS;
```

9.7.6 Windows NT 4.0 Kernel Mode Clients

This binding is used by Kernel Mode clients supporting Windows NT 4.0. Windows NT Kernel Mode device drivers operate in 32-bit protect mode using a flat memory model. In the flat memory model, all of the x86 segments registers contain selectors describing the same four gigabyte address range.

9.7.6.1 Presence Detection

Windows NT 4.0 Kernel Mode clients detect the presence of Card Services by using the service **IoGetDeviceObjectPointer** to look for a device named "\\Device\\PCMCIA$". If successful then

this service returns a pointer to the Card Services Device object. If the request is not sucessful then Card Services is not present.

9.7.6.2 Making Card Services Requests

Windows NT 4.0 Kernel Mode clients make Card Services requests using by placing appropriate values in the parameters and argument packet fields as described below and using the IOCTL instructions:

```
#define FILE_DEVICE_PCCARD    0xc353
#define CS_REQUEST            0x800
#define IOCTL_PCMCIA_CardServices                                    \
                        CTL_CODE(FILE_DEVICE_PCCARD, CS_REQUEST,  \
                                 METHOD_BUFFERED, FILE_ANY_ACCESS)
```

9.7.6.3 Argument Passing

Windows NT 4.0 Kernel Mode clients pass all parameters for Card Services using the following data table:

Offset	Size (8-bit bytes)	Title (input/output)
0	4	*Service* /retcode (Card Services return code)
4	2	*Handle*
6	2	*Reserved*
8	4	*Pointer*
12	4	*ArgLength*
16	n (value of *ArgLength*)	*ArgBuffer*

9.7.6.4 Binding Specific Arguments and Services

All *Pointer* and **ArgPointer** service arguments are 32-bit pointers.

AddSocketServices (32H)

CardServices(**AddSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to be added to those that Card Services is already using. The *Pointer* argument contains the Socket Services *32-bit* entry point. Card Services *makes a CALL* to Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	3	*Three (3) indicating the SSEntry and DataPointer arguments are 32-bit offsets.*
2	DataPointer	4	I	N	*32-bit offset of the Socket Services data area, stored in little-endian format.*

The *Attributes* field must be set to three (3) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are 32-bit offsets.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [ESI] register when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH — *ArgLength* not equal to six (6).

…

RegisterClient (10H)

…

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped.
2	Event Mask	2	I	N	Events to notify client.
4	Client Data	2	I	N	Reference data returned to client in for event notification.
6	Reserved	2	I	ZERO	Reserved (must be reset to zero).
8	Client Data Offset	4	I	N	32-bit offset of client's data area. Placed in [ESI] before calling client callback handler for event notification.
12	Version	2	I	BCD	The *CSLevel* this client expects.

…

ReplaceSocketServices (33H)

CardServices(**ReplaceSocketServices**, *null*, *SSEntry*, *ArgLength*, *ArgPointer*)

This service allows a new Socket Services handler to replace an existing one that Card Services is already using. The new Socket Services implementation must provide functionality that is backward compatible with the Socket Services handler being replaced. The *Pointer* argument contains the Socket Services *32-bit* entry point. Card Services *makes a CALL* to Socket Services at the provided entry point to determine supported hardware.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Base logical socket number
2	NumSockets	2	I	N	Number of sockets to replace
4	Attributes	2	I	3	*Three (3) indicating the SSEntry and DataPointer arguments are 32-bit offsets.*
6	DataPointer	4	I	N	*32-bit offset of Socket Services data area, stored in little-endian format.*

…

The *Attributes* field must be set to three (3) to indicate the Socket Services entry point in the *Pointer* argument and the *DataPointer* fields are 32-bit offsets.

The *DataPointer* field is used to establish data addressability for the Socket Services handler. This value is passed to the Socket Services handler in the [ESI] register when Card Services calls the handler.

…

Return Codes

BAD_ARG_LENGTH — *ArgLength* not equal to ten (10).

…

RequestWindow (21H)

…

The *Attributes* field is bit-mapped. It is defined as follows:

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit / set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	Window below one megabyte (set = true)
Bit 8	Card offsets are window sized (set = true)
Bit 9	Data path width (set = 32 bit / reset = see Bit 3)
Bit 10	Address register (set = expansion ROM / reset = memory Base Address Register)
Bit 11	Pre-fetch (set = prefetchable)
Bit 12	Cache (set = cacheable)
Bits 13 ·· 15	RESERVED (reset to zero)

The *Window below one megabyte* bit indicates that Card Services shall locate the window within the first one megabyte of system address space. This bit is only significant when the *Base* field is reset to zero (0) on entry.

…

9.7.6.5 Client Callback Handler

Windows NT 4.0 Kernel Mode Card Services pass all event notification arguments for client callback handlers at IRQL PASSIVE_LEVEL using the following notification method and data packet:

```
#define CS_CALLBACK     0x802
#define IOCTL_PCMCIA_CallBack                                          \
                        CTL_CODE(FILE_DEVICE_PCCARD, CS_CALLBACK, \
                                METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Offset	Size	Description
0	4	Reserved, reset to zero
4	2	*ClientData*
6	2	Reserved, reset to zero
8	4	ClientOffset
12	4	MTDRegOff
16	4	Reserved
20	2	*Misc* | *Buffer* Lo 16-bits
22	2	*Buffer* Hi 16-bits
24	2	*Info*
26	2	Reserved, reset to zero
28	1	*Socket*
29	1	*Function*
30	2	Reserved, reset to zero
32	2	*Event* / *Status*
34	2	Reserved, reset to zero

Note: All Card Services Callback events occur at IRQL PASSIVE_LEVEL. If the client needs to process the event at a higher level then the client is expected to follow the appropriate steps to raise its IRQL.

9.7.6.6 Media Access Table and MTD Helper Access

The Media Access Table is defined to provide a pointer to the MTDHelper routines of Card Services. In Windows NT 4.0 this pointer is not used, instead an IOCTL is defined below for Kernel Mode MTD Clients use to access the MTD helper routines of Card Services. The placeholder for the MTDHelper pointer has a null value. Kernel Mode Windows NT 4.0 MTD clients use the data packet below to pass parameters to the MTDHelper routines.

```
#define CS_MTDHELPER    0x803
#define IOCTL_PCMCIA_MTD_HELPER                                        \
                CTL_CODE(FILE_DEVICE_PCCARD, CS_MTDHELPER,      \
                        METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Offset	Size	Description
0	4	Reserved, reset to zero
4	2	CardOffsetHi (high 16 bits) | WndHostLo (lo 16 bits)
6	2	WndHostHi (high 16 bits)
8	2	CardOffsetLo (lo 16 bits)
10	10	Reserved
20	1	AccessSpeed | Vpp1 | MTDReadyEnabled
21	1	Attributes | Vpp2
22	2	Reserved
24	2	WindowHandle | Logical Socket
26	2	Reserved
28	2	WindowSize
30	2	Reserved
32	2	MTDHelperSubfunction / RetCode
34	2	Reserved
36	4	Reserved
40	2	ErrorFlag

APPENDIX-E

10. MTD HELPER SERVICE REFERENCE

These services are defined for x86 architecture system DOS and OS/2 environments. Address include SEGMENT16 values for x86 architecture system DOS environments and SELECTOR16 values for x86 architecture system 286 protected mode OS/2 environments.

These services are requested via the *Pointer* in the Media Access Table passed to MTDs. These services should only be used by an MTD during its processing of a read, write, or erase request.

All MTD Helper services pass information in the following registers:

[AX] = MTD Helper Entry Value

All MTD Helper services return [CF] reset to zero if the request was successfully processed. [CF] is set to one if the request was not completed for some reason. [AH] is returned with a value that indicates more detailed information of the request success/failure. This release of the Card Services Interface Specification does not define the values of the return codes.

10.1 MTDModifyWindow (00H)

This service changes the mapping for the window descriptor to the currently specified values. The values in the request packet for socket, window, system base address, and window size must not be changed from the values originally passed to the MTD by **MTDRequestWindow**.

Entry:

[AX] = 00H — MTDModifyWindow
[DX] = WindowHandle (returned by **RequestWindow**)
[BH] = Attributes

	Bit	0	Reserved by Card Services (ignored by MTD)
	Bit	1	Type of Memory (set = attribute, reset = common)
	Bit	2	Reserved (reset to zero)
	Bit	3 .. 4	Prefetchable / Cacheable
			0 = neither prefetchable nor cacheable
			1 = prefetchable but not cacheable
			2 = both prefetchable and cacheable
			3 = Reserved value, do not use
	Bit	5 .. 7	Reserved (reset to zero)

[BL] = Access Speed

	Bit	0 .. 2	Device Speed Code, if Speed Mantissa is zero
	Bit	0 .. 2	Speed Exponent, if Speed Mantissa is non-zero
	Bit	3 .. 6	Speed Mantissa
	Bit	7	Use WAIT, if available

[DI]:[SI] = Card Offset

Exit:

Returns [CF] reset to zero if service completed successfully. [CF] is set to one if the changes to the window mapping could not be made.

The window must be released before the MTD returns from Card Services.

10.2 MTDReleaseWindow (01H)

This service returns to Card Services a window descriptor that an MTD had requested.

Entry:

[AX] = 01H — **MTDReleaseWindow**

[DX] = WindowHandle (returned by **RequestWindow**)

Exit:

Returns [CF] reset to zero if service completed successfully. [CF] is set to one if this window had not been previously requested.

The window descriptor is the same as for **MTDModifyWindow**.

10.3 MTDRequestWindow (02H)

This service returns a window descriptor that an MTD can use to control direct access to memory.

Entry:

[AX] = 02H — **MTDRequestWindow**

[BH] = Attributes

Bit	0	Reserved by Card Services (ignored by MTD)
Bit	1	Type of Memory (set = attribute, reset = common)
Bit	2	Card offset alignment on size boundary (output only)
Bit	3 ·· 4	Prefetchable / Cacheable
		0 = neither prefetchable nor cacheable
		1 = prefetchable but not cacheable
		2 = both prefetchable and cacheable
		3 = Reserved value, do not use
Bit	5 ·· 7	Reserved (reset to zero)

[BL] = Access Speed

Bit	0 ·· 2	Device Speed Code, if Speed Mantissa is zero
Bit	0 ·· 2	Speed Exponent, if Speed Mantissa is non-zero
Bit	3 ·· 6	Speed Mantissa
Bit	7	WAIT required

[CX] = Window size (in 4 KByte units), if zero, largest size available will be returned

[DX] = Logical socket

Exit:

[DX]	=	WindowHandle (returned by **RequestWindow**)
[ES]:[DI]	=	System address for Window
[CX]	=	Window size
[BH]	=	Bit 2 indicates the PC Card offset alignment requirements for PC Card mapping. If reset to zero, the PC Card offset can be any 4 KByte boundary. If set to one, the PC Card offset must be a window size multiple.

Returns [CF] reset to zero if service completed successfully. [CF] is set to one if no windows are available.

The window descriptor is the same as for **MTDModifyWindow**.

10.4 MTDSetVpp (03H)

This service sets the **VPP1** and **VPP2** levels for the socket to the requested values.

Entry:

[AX] = 03H — **MTDSetVpp**

[BL] = **VPP1** Voltage to set in 0.1V increments (0.0 V - 25.5 V)

[BH] = **VPP2** Voltage to set in 0.1V increments (0.0 V - 25.5 V)

[DX] = Logical Socket

Exit:

An MTD must set **VPP1** and **VPP2** back to **VCC** after completing its request. Returns [CF] reset to zero if the service completed successfully and the voltage is stable. [CF] is set to one if the voltage could not be set.

10.5 MTDRDYMask (04H)

This service enables and disables the READY event mask to allow an MTD to avoid generating extraneous READY events in the system.

Entry:

[AX] = 04H — **MTDRDYMask**

[BL] = set to one to enable, reset to zero to disable

Exit:

No exit parameters.

APPENDIX-F

11. MEDIA ACCESS SERVICES REFERENCE

11.1 CardSetAddress

These services are defined for x86 architecture system compatible DOS and OS/2 environments. Address include SEGMENT16 values for x86 architecture system DOS environments and SELECTOR16 values for x86 architecture system 286 protected mode OS/2 environments.

Entry:

[AX]:[DX]		=	PC Card absolute address
[BH]		=	bit mapped attributes
Bit	0	=	attribute memory (set to 1) common memory (reset to 0)
Bit	1	=	read requests (reset to 0) write requests (set to 1)
Bit	2	=	reserved (reset to 0)
Bit	3 ·· 4		Prefetchable / Cacheable 0 = neither prefetchable nor cacheable 1 = prefetchable but not cacheable 2 = both prefetchable and cacheable 3 = Reserved value, do not use
Bit	5 ·· 7	=	reserved (reset to 0)
[BL]		=	access speed
[CX]		=	logical socket

Exit:

[DX] = MAT transfer token value for use by read/write requests

[DS]:[SI] = MAT transfer token value for use by read requests

— or —

[ES]:[DI] = MAT transfer token value for use by write requests

[CX] = maximum number of bytes that can be transferred when using auto-increment addressing before another **CardSetAddress** is required

The processor direction flag is cleared which causes string instructions to increment system addresses.

Auto-increment PC Card memory addressing is turned on.

CardSetAddress must be called before performing any PC Card memory access with the other media access requests. **CardSetAddress** controls the adapter to allow access to PC Card memory via the other media access requests.

11.2 CardSetAutoInc

Entry:

[AX] = reset to 0 turns off auto-increment, set to 1 turns on auto-increment

Exit:

The processor direction flag is cleared which causes string instructions to increment system addresses.

CardSetAutoInc controls the adapter to turn on or off auto-incrementing. If auto-incrementing is turned on, the Card requests with AI at the end requests must be used (see below). If auto-incrementing is turned off, the non Card requests without the AI at the end must be used.

11.3 CardRead(Byte, Word, ByteAI, WordAI)

Entry:

[DX]	=	MAT transfer token value
[DS]:[SI]	=	MAT transfer token value

Exit:

[AX]	=	data read for **ReadWord**
	— or —	
[AL]	=	data read for **ReadByte**
[DX]	=	new MAT transfer token value
[DS]:[SI]	=	new MAT transfer token value

These routines read a byte or word from PC Card memory. The **CardRead** routines with **AI** at the end must be used if auto-incrementing is turned on, and the **CardRead** routines without **AI** at the end must be used if auto-incrementing is turned off.

11.4 CardRead(Words, WordsAI)

Entry:

[CX]	=	number of words to transfer
[DX]	=	MAT transfer token value
[DS]:[SI]	=	MAT transfer token value
[ES]:[DI]	=	pointer to buffer for data read

Exit:

[ES]	=	unchanged
[DI]	=	input [DI] + input [CX]
[CX]	=	number of bytes remaining
[DX]	=	new MAT transfer token value
[DS]:[SI]	=	new MAT transfer token value

CardReadWords and **CardReadWordsAI** read words from PC Card memory into the supplied buffer. The current PC Card memory address must be word aligned. **CardReadWordsAI** can only be used if auto-incrementing is turned on.

11.5 CardWrite(Byte, Word, ByteAI, WordAI)

Entry:

[AX]	=	data to write for **WriteWord**
	— or —	
[AL]	=	data to write for **WriteByte**
[DX]	=	MAT transfer token value
[ES]:[DI]	=	MAT transfer token value

Exit:

[DX]	=	new MAT transfer token value
[ES]:[DI]	=	new MAT transfer token value

These routines write a byte or word to PC Card memory. The **CardWrite** routines followed by **AI** must be used if auto-incrementing is turned on, and the **CardWrite** without **AI** at the end must be used if auto-incrementing is turned off.

11.6 CardWrite(Words, WordsAI)

Entry:

[CX]	=	number of words to write
[DX]	=	MAT transfer token value
[ES]:[DI]	=	MAT transfer token value
[DS]:[SI]	=	pointer to buffer for data to write

Exit:

[DS]	=	unchanged
[SI]	=	input [SI] + input [CX]
[CX]	=	number of bytes remaining
[DX]	=	new MAT transfer token value
[ES]:[DI]	=	new MAT transfer token value

CardWriteWords and **CardWriteWordsAI** writes words to PC Card memory. The current PC Card memory address must be word aligned. **CardWriteWordsAI** can only be used if auto-incrementing is turned on.

11.7 CardCompare(Byte, ByteAI)

Entry:

[AL] = byte to compare
[DX] = magic value
[ES]:[DI] = magic value

Exit:

[DX] = new magic value
[ES]:[DI] = new magic value
CF = reset to zero (0) if compare was successful

11.8 CardCompare(Words, WordsAI)

Entry:

[CX] = number of words to compare
[DX] = magic value
[ES]:[DI] = magic value
[DS]:[SI] = pointer to buffer for data to compare

Exit:

[DS] = unchanged
[SI] = input [SI] + input [CX]
[CX] = number of bytes remaining
[DX] = new magic value
[ES]:[DI] = new magic value
CF = reset to zero (0) if compare was successful

CardCompareWords and **CardCompareWordsAI** compare words on the PC Card with words in the supplied buffer. The current PC Card memory address must be word aligned.
CardCompareWordsAI can only be used if auto-incrementing is turned on.

APPENDIX-G

12. ARGUMENT USAGE REFERENCE

This table indicates which arguments are used for each Card Services request. The Handle argument indicates input value followed by output value (Input/Output). If both values are the same, one value is listed. A ✓ indicates that the argument is used for the listed request. No entry for an argument indicates that the request does not use that argument. The value of the *Status* argument is returned by Card Services to the requesting client.

Request	Service	Handle	Pointer	ArgLen	ArgPtr	Status
AccessConfigurationRegister	✓			✓	✓	✓
AddSocketServices	✓		entry	✓	✓	✓
AdjustResourceInfo	✓			✓	✓	✓
CheckEraseQueue	✓	Queue		✓		✓
CloseMemory	✓	Memory/–				✓
CopyMemory	✓	Memory		✓	✓	✓
DeregisterClient	✓	Client/–				✓
DeregisterEraseQueue	✓	Queue/–				✓
GetCardServicesInfo	✓			✓	✓	✓
GetClientInfo	✓	Client		✓	✓	✓
GetConfigurationInfo	✓	–/Client		✓	✓	✓
GetEventMask	✓	Client		✓	✓	✓
GetFirstClient	✓	–/Client		✓	✓	✓
GetFirstPartition	✓			✓	✓	✓
GetFirstRegion	✓	–/Client		✓	✓	✓
GetFirstTuple	✓			✓	✓	✓
GetNextClient	✓	Client/Client		✓	✓	✓
GetNextPartition	✓			✓	✓	✓
GetNextRegion	✓	–/Client		✓	✓	✓
GetNextTuple	✓			✓	✓	✓
GetStatus	✓			✓	✓	✓
GetTupleData	✓			✓	✓	✓
MapLogSocket	✓			✓	✓	✓
MapLogWindow	✓	Window		✓	✓	✓
MapMemPage	✓	Window		✓	✓	✓
MapPhySocket	✓			✓	✓	✓
MapPhyWindow	✓	–/Window		✓	✓	✓
ModifyConfiguration	✓	Client		✓	✓	✓
ModifyWindow	✓	Window		✓	✓	✓
OpenMemory	✓	Client/Memory		✓	✓	✓

Request	Service	Handle	Pointer	ArgLen	ArgPtr	Status
ReadMemory	✓	Memory	buffer	✓	✓	✓
RegisterClient	✓	–/Client	entry	✓	✓	✓
RegisterEraseQueue	✓	Client/Queue	queuehead			✓
RegisterMTD	✓	Client		✓	✓	✓
RegisterTimer	✓	Client/Timer		✓	✓	✓
ReleaseConfiguration	✓	Client		✓	✓	✓
ReleaseExclusive	✓	Client		✓	✓	✓
ReleaseIO	✓	Client		✓	✓	✓
ReleaseIRQ	✓	Client		✓	✓	✓
ReleaseSocketMask	✓	Client		✓	✓	✓
ReleaseWindow	✓	Window/–				✓
ReplaceSocketServices	✓		entry	✓	✓	✓
RequestConfiguration	✓	Client		✓	✓	✓
RequestExclusive	✓	Client		✓	✓	✓
RequestIO	✓	Client		✓	✓	✓
RequestIRQ	✓	Client	ISREntry/-	✓	✓	✓
RequestSocketMask	✓	Client		✓	✓	✓
RequestWindow	✓	Client/Window		✓	✓	✓
ResetFunction	✓	Client		✓	✓	✓
ReturnSSEntry	✓		–/entry	✓	✓	✓
SetEventMask	✓	Client		✓	✓	✓
SetRegion	✓			✓	✓	✓
WriteMemory	✓	Memory	buffer	✓	✓	✓
ValidateCIS	✓			✓	✓	✓

APPENDIX-H

13. CLIENT CALLBACK ARGUMENT USAGE

This table indicates which arguments are used for each Card Services callback procedure. If both values are the same, one value is listed. A ✓ indicates that the argument is used for the listed request. No entry for an argument indicates that the request does not use that argument. The value of the *Status* argument is returned by a client to Card Services.

Callback	Status	Service	Socket	Info	MTD Request	Buffer	Misc	Client Data
Insertion								
CARD_INSERTION		✓	✓				Client Handle	✓
Registration Completion								
REGISTRATION_COMPLETE		✓					Client Handle	✓
Status Change								
BATTERY_DEAD		✓	✓					✓
BATTERY_LOW		✓	✓					✓
CARD_LOCK		✓	✓					✓
CARD_UNLOCK		✓	✓					✓
CARD_READY		✓	✓					✓
CARD_REMOVAL		✓	✓					✓
PM_SUSPEND	✓	✓		✓		✓		✓
PM_RESUME		✓		✓			Mode	✓
WRITE_PROTECT		✓	✓	✓				✓
REQUEST_ATTENTION		✓	✓					✓
Ejection/Insertion Requests								
EJECTION_REQUEST	✓	✓	✓					✓
EJECTION_COMPLETE		✓	✓					✓
INSERTION_REQUEST	✓	✓	✓					✓
INSERTION_COMPLETE		✓	✓					✓
Exclusive								
EXCLUSIVE_REQUEST	✓	✓	✓					✓
EXCLUSIVE_COMPLETE		✓	✓	✓				✓
RESET								
RESET_REQUEST	✓	✓	✓					✓
RESET_PHYSICAL		✓	✓					✓
CARD_RESET		✓	✓	✓				✓
RESET_COMPLETE		✓	✓	✓				✓

Callback	Status	Service	Socket	Info	MTD Request	Buffer	Misc	Client Data
Client Information								
CLIENT_INFO	✓	✓				✓		✓
Erase Completion								
ERASE_COMPLETE		✓	✓	✓			Erase Que Handle	✓
MTD Request								
MTD_REQUEST	✓	✓	✓		✓	✓		✓
Timer								
TIMER_EXPIRED		✓					Timer Handle	✓
New Socket Services								
SS_UPDATED		✓	✓	✓				✓

APPENDIX-I

14. OS CRITICAL SECTION HANDLING

Card Services is designed with the intention that a client should never find a situation in which a Card Services call is failed because Card Services is not enterable. However, in a DOS system a TSR or application which performs Card Services calls can be in the middle of a Card Services call and the system can switch to another task because it is running under Windows, Task Swapper, DesqView or a similar environment. The new task can then attempt to use the file system client which in turn uses Card Services. This causes a critical error (Abort, Retry, Ignore, Fail) or other catastrophic error to be returned because the file system will find Card Services cannot be entered, since the previous task is still in Card Services.

For clients such as file systems (block / character device drivers, etc.), this is not a problem because access to Card Services is synchronized through MS-DOS's own critical section handling, and the task-switching environments have all been designed to not switch away from a task during the middle of an MS-DOS function call. However, TSR's and generalized applications do not have the same synchronization, and it is therefore necessary for any TSR or application which makes Card Services calls to encapsulate ALL such calls within the Enter / Leave Critical Section APIs appropriate to the detected task switching environment.

Clients in task-switching environments should use the Critical Section handler APIs appropriate to such environments around EVERY call to Card Services. Information about these APIs can be found in following:

Windows:	The Windows Device Development Kit (DDK), Virtual Device Adaptation Guide, Appendix D — Windows INT 2F API.
MS-DOS Task Switcher:	The MS-DOS 5.0 Programmer's Reference, Section 7.10, Task Swapper Reference, and INT 2F Function 4BXXH.
DesqView:	The DesqView User's Manual, Appendix J contains assembly code fragments to allow the creation of DesqView aware applications, including the necessary API calls.

This Page Intentionally Left Blank