PC CARD STANDARD

Volume 6

Socket Services Specification

REVISION HISTORY

Date	Socket Services Specification Version	PC Card Standard Release	Revisions
09/91	1.01		Initial release as independent specification
11/92	2.0	PCMCIA 2.01	Reformatted Added PC-Compatible Bindings
07/93	2.1	PCMCIA 2.1/JEIDA 4.2	Editorial corrections
02/95	5.0	February 1995 (5.0) Release	Added support for CardBus PC Cards
03/95	N/A	March 1995 (5.01) Update	None
05/95	N/A	May 1995 (5.02) Update	None
11/95	5.1	November 1995 (5.1) Update	Added Custom Interface Support Editorial Corrections
05/96	N/A	May 1996 (5.2) Update	None
03/97	6.0	6.0 Release	Added Hot Dock/Undock Support Editorial Corrections
04/98	6.1	6.1 Update	Added Packet Interface
02/99	7.0	7.0 Release	None
03/00	7.1	7.1 Update	None
11/00	7.2	7.2 Update	None
04/01	8.0	8.0 Release	None

CONTENTS

TABLES

1. INTRODUCTION

1.1 Purpose

This document describes the software interface provided by PC Card Standard Socket Services. This interface provides a hardware independent method of managing PC Card sockets in a host system.

1.2 Scope

This document is intended to provide enough information to software developers to utilize PC Card sockets in a host system without any knowledge of how the actual hardware performs the desired services. It is also intended to provide enough information for an implementer to create a Socket Services handler for a particular adapter.

1.3 Related Documents

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***

This Page Intentionally Left Blank

2. OVERVIEW

Socket Services is the lowest layer in a multi-layer architecture that manages resources on PC Card Standard compatible memory and I/O cards (collectively known as PC Cards). Socket Services provides a universal software interface to the hardware that controls sockets for PC Cards. It masks the details of the hardware used to implement these sockets, allowing higher-level software to be developed which is able to control and utilize PC Cards without any knowledge of the actual hardware interface.

Software layers above Socket Services provide additional capabilities. Immediately above Socket Services is Card Services which arbitrates the use of Socket Services resources. Card Services is responsible for taking requests from multiple processes and sharing the resources provided by Socket Services among these processes. In this manner, Card Services may actually provide the same hardware to different processes allowing the use of the hardware to be time-multiplexed.

For example, if a BPB–FAT partition and a Flash File System partition both reside on a Flash card, Card Services might provide the same windows into the host system memory address space for both of the device drivers involved in accessing those partitions. Card Services is responsible for handling overlapping requests, ensuring that the appropriate partition is available at the right time.

Socket Services approaches the handling of the hardware it manages by addressing it as a number of objects with different areas of functionality. Adapters are the hardware that connects a host system's bus to PC Card sockets. Host systems may have more than one adapter. Socket Services reports the number of sockets, windows and EDC generators provided by each adapter installed. Adapter power consumption and status change reporting may be controlled separately for each adapter.

An adapter may have one or more sockets. Sockets are receptacles for PC Cards. Socket Services describes the characteristics of each socket and allows socket resources to be manipulated and current settings determined.

Socket Services also provides services to deal with PC Cards. These services report on current card status, allow data to be read and/or written on 16-bit PC Cards which are not mapped into system memory address space, and allow configuration space to be read and/or written on CardBus PC Cards.

For performance reasons, it is often beneficial to map PC Cards into host system memory or I/O address space. (XIP requires the ability to map PC Card memory arrays into system memory address space.) Adapters may or may not provide this capability. An area of PC Card memory and/or I/O address space is mapped into a corresponding host system area through a window. From the Socket Services perspective, there are three types of hardware that may be involved in mapping a PC Card's address space into a host system's address space.

Adapters for 16-bit PC Cards have windows on the adapter that map PC Card address space into the host system's address space. These windows map memory and/or I/O address space. Devices on a 16-bit PC Card are always located at the same PC Card address. The area of a PC Card mapped into a host system's address space is determined by a combination of adapter decoding and PC Card decoding.

Adapters for CardBus PC Cards do not perform any mapping of PC Card address space into a host system's address space. Base Address Registers on the PC Card itself are programmed to decode host system addresses directly.

If an adapter also acts as a bridge to another host system bus, it may have bridge windows. Bridge windows are used to route a range of host system addresses across the bridge to a PC Card. Bridge

windows are controlled separately from the windows on a 16-bit PC Card adapter or the Base Address Registers on a CardBus PC Card. If an adapter uses bridge windows, the address ranges routed by the bridge windows must include the ranges used by 16-bit PC Card adapter windows or the ranges programmed into the Base Address Registers on a CardBus PC Card.

3. FUNCTIONAL DESCRIPTION

3.1 System Architecture

Socket Services is a software interface to the hardware used to manage PC Card sockets in a host system. Above Socket Services, an operating system-specific layer known as Card Services virtualizes Socket Services to allow it to be shared by multiple processes. These processes may include such things as eXecute-In-Place (XIP), Flash File System (FFS), and other types of device drivers.

Socket Services provides only the lowest level access to PC Cards. For example, Socket Services allows the 16-bit PC Card attribute memory space to be read, but it does not interpret the Card Information Structure (CIS).

Socket Services is invoked in a platform dependent manner. All service arguments are passed to Socket Services in a binding specific fashion. Status of a Socket Services request is returned in the status argument. (See ***Appendix-C, 9.*** *Socket Services Bindings*.) Using functional notation, a Socket Services request generically can be considered as:

```
status = Service(arg1, arg2 ...)
```

While this notation resembles a C language function call, Socket Services is implemented in an appropriate manner for its environment. For example, on an x86 architecture platform a ROM BIOS Socket Services interface is handled through Interrupt 1AH with services based at 80H. A client simply sets the host processor's registers for the service desired and executes the Socket Services software interrupt. Status is returned using the Carry flag ([CF]) and registers specific to the service invoked.

Special handling is required to be able to write many types of memory cards. It is not feasible to attempt to include all the necessary handlers within Socket Services for all the possible types of write/erase routines. Handling of technology-specific write requirements is intended to be performed by a software layer above Socket Services. Socket Services provides access to the hardware for these card technology routines.

3.2 Initialization

Socket Services is internally initialized during installation and no specific installation is required by the client before making service requests. It is expected the client of Socket Services will check the Socket Services *Compliance* to determine the level of service available. (See ***5.3.12 GetSSInfo [BOTH]***.)

3.3 Configuration

The next step is to enumerate the capabilities of the implementation. This entails determining the number of adapters installed, how many sockets, bridge and 16-bit PC Card windows are supported by each adapter, and exploring the power management and indicators available for each adapter.

As noted above, it is expected that Socket Services is virtualized by Card Services. Above Card Services are device drivers for different types of PC Cards. These drivers map PC Cards into system I/O and/or memory space to implement their functions. Multiple drivers may share PC Cards and sockets and may even share windows. Card Services arbitrates requests for Socket Services resources and is responsible for preserving any state information required to share these resources.

3.4 Status Change Notification

A Socket Services client may desire notification when a status change occurs. Status changes include, but are not limited to, the following: card removal or insertion, battery low or dead, and **READY** changes. Socket Services supports steering and enabling status change interrupts from an adapter. A client installs a status change interrupt handler on the host interrupt level selected to receive such interrupts. A client may choose to poll for changes in socket and card status.

When an adapter configured for status change interrupts detects a status change, it generates an interrupt which invokes the client's status callback handler. This handler uses the Socket Services **AcknowledgeInterrupt** service to determine which socket or sockets experienced the status change. It records this information and completes the hardware interrupt processing. Later, during background processing, the client notes which sockets require attention and uses the **GetStatus** service to determine current PC Card and socket state. This state is used to determine what action should be taken by the client. Status change interrupt handling is provided by Card Services. (See the ***Card Services Specification***.)

3.5 Power Management

The Socket Services interface provides controls for conserving adapter power. Two power conservation modes are provided: reduced with all state information maintained and reduced without state information being maintained. These levels are established with the **SetAdapter** service.

Socket Services may also be used to manage power to PC Card sockets. Independent controls and levels are provided for **VCC**, **VPP1** and **VPP2**. Since available power levels are generally limited, Socket Services provides a list of supported levels and then allows power adjustment based on an index into that list. Power management is performed at the socket level. How Socket Services resolves power management requests in hardware implementations that only allow control of power at the adapter level is vendor specific. Socket Services reports the level of power management control available through the **InquireAdapter** service.

3.6 Docking

Whether or not Socket Services is dynamically loaded (or unloaded) there is a general sequence of things that Socket Services needs to perform in order to handle dock events. Considering all possible dock scenarios Socket Services really is performing one of three actions: add support (dock where new controllers are present requiring new Socket Services handlers), remove support (undock where controllers are gone requiring removal of Socket Services handlers) or change / replace support (either dock or undock using same socket services instance). This leads to the following sequences for communications between Socket Services and Card Services:

I. Replace Support

   A. Socket Services issues **ReplaceSocketServices** to Card Services w / Base log. Socket # (obtained via **MapPhyLogSocket**) and number of sockets to replace. Until Socket Services receives **GetSetPriorHandler** or Card Services returns from **ReplaceSocketServices** this Socket Services rejects any request (except **GetSSInfo**, see below for more info) w / BUSY return code.

   B. Upon receipt of **GetSetPriorHandler**

      1. If previous is NULL then return w / adapter 0; else,

      2. Add itself as supporting next adapter (if any such adapter exists that needs support else may take steps to remove itself from memory if environment supports this).

C. Receives return from **ReplaceSocketServices** request.

D. Receives and processes normal “initialization” requests from Card Services.

II. Add Support

A. Socket Services issues **AddSocketServices** to Card Services

B. Socket Services receives **GetSetPriorHandler** and before returning numbers his adapter (via GetSSInfo) and return

C. Returns proper data to **GetSSInfo**

D. Receives return from **AddSocketServices**

E. Receives and processes normal “initialization” requests from Card Services.

III. Remove Support

A. Use same logic flow as Replace Support except return zero (0) supported adapters for **GetSSInfo** request.

NOTE: The **GetSetPriorHandler** request is used by Card Services implementations that expect Socket Services handlers to track the chain of handlers. Some Card Services implementations will track the handlers themselves and in this situation Socket Services may not receive any **GetSetPriorHandler** requests during processing of dock events.

3.7 Overview of Services

3.7.1 Non-specific Service

There is one Socket Services service which applies to the interface in general and not to any objects manipulated by the interface. It is:

GetAdapterCount

3.7.2 Adapter Services

Socket Services addresses adapters with the following services:

AcknowledgeInterrupt
GetSetPriorHandler
GetSetSSAddr
GetAccessOffsets
GetAdapter
GetSSInfo
GetVendorInfo
InquireAdapter
SetAdapter
VendorSpecific

3.7.3 Socket Services

Socket Services addresses sockets with the following services:

GetSocket
GetStatus
InquireSocket
ReSetSocket
SetSocket
AccessConfigurationSpace

3.7.4 Window Services

Socket Services addresses windows with the following services:

GetBridgeWindow
GetPage
GetWindow
InquireWindow
SetBridgeWindow
SetPage
SetWindow
InquireBridgeWindow

WARNING:

Windows which map 16-bit PC Cards into host system memory address space may have one or more pages. If a Window contains multiple pages, each page must be 16 KBytes and windows must be sized as a multiple of the 16 KByte page size.

3.7.5 Error Detection and Correction Services

Adapters and / or Sockets may optionally provide error detection and correction support. The following services handle EDC capabilities:

GetEDC
InquireEDC
PauseEDC
ReadEDC
ResumeEDC
SetEDC
StartEDC
StopEDC

3.7.6 Status Change Handling

Socket Services provides for asynchronous notification when a socket's status changes. Each adapter may provide a hardware interrupt when there is a status change. This interrupt is processed by a handler installed by the Socket Services client.

While only one interrupt per adapter is anticipated, the Socket Services interface allows status changes to be masked on a per socket basis. Masking must be performed in hardware since the hardware interrupt is handled directly by the Socket Services client.

If status change interrupts are supported, each Socket Services client determines which interrupt it uses for status changes based on the set of supported interrupts reported by **InquireAdapter**. A Socket Services client may enable or disable this capability and may steer the interrupt to a supported host interrupt level.

3.7.7 Reserved Services

Depending on the binding, some Socket Services service codes may be reserved for historical reasons and should not be used. If a client uses one of these service codes, an implementation should return BAD_SERVICE.

4. ASSUMPTIONS AND CONSTRAINTS

4.1 ROM Located

The Socket Services interface is intended to allow the handler to be located in ROM on a host platform. To promote this capability, the use of RAM to store status and / or state information is minimized.

4.2 Hardware Implementation

While the Socket Services interface has been developed to mask the details of the actual hardware used to implement PC Card sockets, some hardware implementations do provide advantages. As noted above, Socket Services is intended to be located in ROM. This requires that the amount of RAM used by Socket Services is as small as possible. Using hardware registers which are read / write, rather than write-only, allows state information to be determined by reading the hardware and not by maintaining RAM-based copies of values previously written to write-only registers.

Another area where hardware implementation can simplify or complicate Socket Services is status reporting. Hardware registers that are automatically reset when read force Socket Services to keep RAM-based copies of values read to ensure status information is not lost when read by routines not interested in the particular status returned. On the other hand, if status registers require explicit resets, status information is maintained until acknowledged by the appropriate software routine. This provides a positive acknowledgment that the status condition has been noted and resolved. For the same reason, if multiple status bits reside in the same register, they must be able to be reset on an individual basis.

4.3 Adapters Supported

The Socket Services interface allows multiple adapters containing one or more PC Card sockets. The actual number of adapters supported is limited by several factors. These include: the specifics of the platform binding, the constraints imposed by locating Socket Services in ROM, and a particular vendor's implementation.

Adapters are numbered from zero (0) to the maximum (one less than the number of adapters installed as returned by **GetAdapterCount**).

4.4 Sockets Supported

The Socket Services interface allows multiple PC Card sockets per adapter. The maximum number of sockets an adapter can support is primarily limited by the fact that a bit-map of assignable sockets is returned by the **InquireWindow** service. As with adapters, the constraints imposed by locating Socket Services in ROM may impose a smaller limit on the number of sockets supported. An adapter may support any number of sockets, from one to the theoretical maximum imposed by the number of bits in the field used to return the bit-map of assignable sockets. If a system has more than one adapter, each adapter may support a different number of sockets.

Sockets are numbered from zero (0) to one less than the number on the adapter (as returned by **InquireAdapter**). The maximum number of sockets that may be supported depends on the Socket Services binding.

4.5 Windows Supported

The Socket Services interface is designed without any assumptions about how or whether PC Cards are mapped into the host's I/O or memory space. This requires a mechanism to indicate which windows can be mapped to a particular socket. Socket Services uses a bit-map to return this information as described in the **InquireWindow** and **InquireBridgeWindow services**.

There are two types of windows managed by *Socket Services*. The **InquireAdapter** service returns the number of both types of windows on the adapter.

The first window type supports memory or I/O accesses to a 16-bit PC Card. Hardware on the adapter performs initial decoding of a host system access. If this access is within the address range of the window, the window hardware asserts the Card Enable signal to the PC Card socket. This informs the PC Card that it needs to perform further decoding to respond to the access. 16-bit PC Card address decoding is a combination of the adapter and PC Card hardware.

The **InquireWindow** service returns the characteristics of 16-bit PC Card windows. The current configuration of these windows is returned by the **GetWindow** service and the window is configured using the **SetWindow** service.

CardBus PC Cards do not use this first type of window. CardBus PC Cards perform all address decoding using Base Address Registers on the card that have been programmed for a specific host system address range.

The second type of window managed by *Socket Services* is used only when the PC Card adapter is a bridge to a host system bus. Bridge windows route a range of host system memory or I/O accesses to a PC Card socket. 16-bit PC Card address decoding is a combination of window hardware on the adapter (as noted above) and decoding on the card. CardBus PC Card address decoding is performed entirely by the card based on value programmed into the card's Base Address Register(s). The characteristics of a bridge window are returned by **InquireBridgeWindow**. The current configuration is returned by **GetBridgeWindow** and a bridge window is configured using **SetBridgeWindow**.

A particular implementation may choose not to provide any mapping of 16-bit PC Cards into the host system's I/O or memory space. In this case the number of windows supported by a particular adapter should be set to zero (0).

If a hardware implementation provides a single window per socket, the **InquireAdapter** service indicates the same value as the number of sockets supported by the adapter. If a hardware implementation allows any of an adapter's windows to be mapped to any of its sockets, the number of windows available should be returned. (Do not multiply the number of windows by the number of sockets, in this case, just use the number of individual windows on the adapter.)

There is no requirement that hardware allow a window to be mapped to more than one socket. However, the Socket Services interface does not prevent a window from being assignable to more than one socket. It is assumed that a window is mapped to only one socket at a time. A window may be shared between sockets if it is specifically remapped between uses by the Socket Services client.

Higher-level software is expected to evaluate the window descriptions returned by Socket Services to determine capabilities available. Socket Services shall fail requests that are invalid, such as attempting to map a window to an unsupported socket. As noted above, Socket Services does not consider it an error to map a window that has been previously mapped. Window mapping state information must be preserved by the Socket Services client. While Socket Services does not preserve prior state information, the client may request current state information. In this case, the client uses the various **'Get'** services prior to setting new state with the various **'Set'** services.

Windows are numbered from zero (0) to one less than the number on the adapter (as returned by **InquireAdapter**). The maximum number of windows that may be supported depends on the Socket Services binding.

4.6 EDC Generators

Error Detection Code generators are optional. EDC generators are numbered from zero (0) to one less than the number on the adapter (as returned by **InquireAdapter**). The maximum number of EDC generators that may be supported depends on the Socket Services binding.

4.7 Power Management and Indicators

Power management and indicators may be available on a per adapter or per socket basis. To provide a consistent interface, Socket Services provides access to these services on a socket basis. It is expected that a hardware implementation that only provides power management and / or indicator control at the adapter level shall provide a Socket Services handler that manages those resources for the entire adapter based on requests to individual sockets.

Socket Services does indicate whether power management and indicator control is performed at the adapter or socket level. However, by providing only one control point (the socket), a client of Socket Services is not required to provide two types of controlling routines.

4.8 Calling Conventions

The Socket Services interface uses a common set of conventions for all services. They are described below.

4.8.1 Reserved Fields

Any reserved fields or undefined bits in entry fields may be ignored by a handler implementing this release of Socket Services. However, reserved fields and undefined bits should be reset to zero before invoking a Socket Services service because future releases of Socket Services may define them. Future releases will use the reset value for behavior compliant with this release of Socket Services.

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

4.8.2 Register Usage

The use of registers to pass arguments and return status is specific to the binding defined for the host platform. See the appropriate binding for register usage conventions. Please note that conventions are guidelines used to develop the service interfaces and exceptions have been made in specific cases.

Whenever possible the interface preserves the contents of all arguments unless they are used to return information. For bit-mapped fields, bits within a field (or register) are numbered beginning with zero. The location of Bit 0 in a field is binding specific.

4.9 Socket Services Generally Not Re-entrant

Except for the **AcknowledgeInterrupt** service, Socket Services is not intended to be re-entrant. Attempting any other Socket Services service while there is a thread of execution within Socket Services may be invalid depending on the implementation. Should a client attempt to re-enter Socket Services for any request other than **AcknowledgeInterrrupt**, the request may be failed returning BUSY.

4.10 Critical Areas and Disabled Interrupts

Socket Services handlers should strive to minimize the amount of time interrupts are disabled. However, a Socket Services handler NEVER enables interrupts during **AcknowledgeInterrupt** processing.

4.11 Request Rejection

Socket Services validates all parameters before changing any hardware. A client is assured that if a request is rejected due to an invalid parameter, no hardware changes have been made based on the request.

5. PROGRAM INTERFACE

5.1 Presence Detection

The presence of Socket Services is determined by performing the **GetAdapterCount** request. (See ***5.3.5 GetAdapterCount [BOTH]***.) If this service returns with a **RETCODE** other than SUCCESS, the client may assume that services provided by Socket Services are not available. If it returns SUCCESS in the *status* field and the ASCII characters 'SS' in the *Signature* field, at least one Socket Services handler is installed.

5.2 Data Types

Socket Services uses a number of defined data types to describe arguments and return codes. Each Socket Services binding describes how these types are defined within the binding. The data types used to describe service parameters are listed below:

Data Type	Meaning
ADAPTER	Specifies a physical adapter. Ranges from zero to one less than the number of adapters present in the host system as reported by **GetAdapterCount.**
BASE	Describes the base address of a window used to map a PC Card's address space into a host system's address space.
BCD	Binary Coded Decimal value. For example, 0221H represents 2.21.
BYTE	An 8-bit quantity.
COUNT	Number of objects of the specified type.
DWORD	A 32-bit quantity.
Double Word	A 32-bit quantity, see DWORD.
EDC	Specifies an Error Detection Code generator. Ranges from zero to one less than the number of EDC generators on the adapter as reported by **InquireAdapter**.
FLAGS8	Bit-mapped field with up to 8 significant bits.
FLAGS16	Bit-mapped field with up to 16 significant bits.
FLAGS32	Bit-mapped field with up to 32 significant bits.
IRQ	IRQ status or control. Includes host system IRQ level, active level (low or high) and state (enabled or disabled).
OFFSET	An address in any of the PC Card's memory address spaces. For a 16-bit PC Card this includes both the attribute and common memory plane. For a CardBus PC Card this includes configuration space, any of the 6 possible memory spaces and the expansion ROM.
PAGE	Subdivision of a window. Ranges from zero to one less than the number of pages in a window. Windows may be a single page of any size or multiple pages of 16 KBytes.
PTR	A pointer to a location in system memory.
PWRENTRY	An entry in an array of items returned by **InquireAdapter**. Describes a specific power level and its valid signals (**VCC**, **VPP1** and **VPP2**).
PWRINDEX	Index into power management table. Ranges from zero to one less than the number of power levels in the array of PWRENTRY items returned by **InquireAdapter**.
RETCODE	Value returned by Socket Services when a service has been processed.
SIGNATURE	Two ASCII characters ('SS') used to validate a Socket Services handler is installed.
SIZE	The size of a window. Memory and I/O windows may use different units.
SKTBITS	Bit-map of valid sockets to which window or EDC generator may be assigned.
SOCKET	Specifies a physical socket. Ranges from zero to one less than the number of sockets on an adapter as reported by **InquireAdapter**.

Data Type	Meaning
SPEED	Encoded value representing a memory window access speed. (See the ***Metaformat Specification***.)
WINDOW	Specifies a physical window. Ranges from zero to one less than the number of windows on an adapter as reported by **InquireAdapter**.
WORD	A 16-bit quantity.

5.3 Service Descriptions

Each Socket Services service is described in detail on the following pages. The descriptions are intended to be processor and operating system independent. Specific bindings for particular environments are specified in appendices to this specification.

Service names are constructed from an action verb and a noun. The noun identifies the type of object being manipulated. If the verb is **Inquire** the capabilities of the object are returned by the service. If the verb is **Get** the current configuration of the item is returned. If the verb is **Set** the item is configured by the service.

The following notation conventions are used:

Convention	Meaning
bold	Bold type is used for keywords. For example, the names of services, data types, structures, and macros. These names are spelled exactly as they should appear in source programs. Defined data types are also in uppercase.
italics	Italic type is used to indicate the name of an argument. The name must be replaced by an actual argument. Italics are also used to show emphasis in text.
`monospace`	Monospace type is used for example program code fragments.
ALL_UPPER	Type in all uppercase is used to indicate a constant value with the exception that defined data types are all uppercase and bold.

The description of each service begins with a heading that contains the service name as described above on the left and an indicator on the right as follows:

[PC16]	Service only applies to 16-bit PC Cards
[PC32]	Service only applies to CardBus PC Cards
[BOTH]	Service applies to both 16-bit PC Cards and CardBus PC Cards

5.3.1 AccessConfigurationSpace [PC32]

RETCODE = AccessConfigurationSpace (*Adapter, Socket, Function, Action, Location, Data*)

ADAPTER *Adapter*;
SOCKET *Socket*;
BYTE *Function*;
FLAGS8 *Action*;
OFFSET *Location*;
FLAGS32 *Data*;

Provides an interface for Card Services to read and write values in CardBus configuration space. This is used to support the Card Services **AccessConfigurationRegister** service as well as to allow Card Services to allocate windows for CardBus PC Card functions by writing to the function's Base Address Registers.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.
Function	I	Specifies the card function whose configuration space is to be accessed.
Action	I	The READ (00H) or WRITE (01H) operation to be performed.
Location	I	The offset into the function's configuration space where the data is to be obtained or written. This value must be aligned on a four-byte boundary.
Data	I/O	If Action is READ, this field returns the data read. If Action is WRITE, this field contains the data to be written. This field always contains a DWORD value.

Return Codes

SUCCESS	Operation was successful
BAD_ADAPTER	Specified *Adapter* is invalid
BAD_ATTRIBUTE	Specified *Action* is not READ or WRITE.
BAD_OFFSET	*Location* is beyond the legal configuration space or is not aligned on a four byte boundary.
BAD_SOCKET	Specified *Socket* and/or *Function* is invalid

5.3.2 AcknowledgeInterrupt [BOTH]

RETCODE = AcknowledgeInterrupt (*Adapter, Sockets*)
ADAPTER *Adapter*;
SKTBITS *Sockets*;

The **AcknowledgeInterrupt** service returns information about which socket or sockets on the adapter specified by the input parameters has experienced a change in status.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Sockets	O	Returns a bit-map representing the sockets which have experienced a status change, e.g. 0021H indicates sockets 0 and 5.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid

Comments

A Socket Services client enables status change interrupts from adapter hardware with the **SetAdapter** service. The client is responsible for installing an interrupt handler on the appropriate vector. Specific events are masked or unmasked on a per socket basis using the **SetSocket** service. When a status change occurs, the handler installed by the client receives control. For each adapter capable of generating that interrupt, the interrupt handler makes an **AcknowledgeInterrupt** request.

The **AcknowledgeInterrupt** request allows Socket Services to prepare the adapter hardware for generating another interrupt if another status change occurs. Socket Services also informs the client which socket or sockets have experienced a status change. Socket Services must preserve state information relating to the cause of the status change interrupt if it is not preserved by the adapter hardware. This information will later be requested with the **GetStatus** service.

After polling all possible adapters with the **AcknowledgeInterrupt** request, the client's interrupt handler prepares the host system for another status change interrupt for the adapter. Some time later, outside of the hardware interrupt handler, the client polls Socket Services for new socket state using **GetStatus**. This service returns a combination of socket and card state information.

By separating the acknowledgment of the interrupt from the retrieval of specific socket and card status, the client may reduce the amount of RAM required to store state information. The client may elect to recover state information only when it is able to fully process the information. In this manner, a client only needs to evaluate complete state information for one socket at a time.

Note: Since adapters may share a status change interrupt, it is possible for this service to be called even if no status change has occurred on the adapter specified. In this case, Socket Services returns indicating success with all bits in *Sockets* reset to zero (0).

WARNING:

AcknowledgeInterrupt *takes place within the status change hardware interrupt. Socket Services must not enable interrupts at any time during the processing of an* ***AnknowledgeInterrupt*** *request.*

See Also **GetStatus**

5.3.3 GetAccessOffsets [PC16]

RETCODE = GetAccessOffsets (*Adapter, Mode, NumDesired, pBuffer, NumAvail*)

ADAPTER *Adapter*;
BYTE *Mode*;
COUNT *NumDesired*;
PTR *pBuffer*;
COUNT *NumAvail*;

The **GetAccessOffsets** service fills the buffer pointed to by *pBuffer* with an array of offsets for low-level, adapter-specific, optimized PC Card access routines for adapters using register-based (I/O port) access to PC Card memory address space. Adapters which access PC Card memory address space through windows mapped into host system memory address space do not support this service.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Mode	I	Specifies the processor mode. This is specific to the type of host platform. See the platform-specific binding for additional detail.
NumDesired	I	Specifies the number of access offsets desired. Indirectly specifies the size of the client-supplied buffer.
pBuffer	I	A pointer to a client-supplied buffer for the array of access offsets. *NumDesired* specifies the number of entries that will fit in the buffer. The offsets are specific to the type of host platform. See the platform-specific bindings for additional details.
NumAvail	O	Returns the number of access offsets supported by this Socket Services handler for the specified adapter.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SERVICE	if request is not supported
BAD_MODE	if *Mode* is not supported

Comments

All of these offsets are in the Socket Services code segment. All sockets on an adapter must use the same entry points for a mode. However, these offsets may vary depending upon the mode specified.

A client uses the returned values to create the MAT passed to MTDs which allows these routines to be called in a manner appropriate to the mode in which they will be used.

There is no requirement that an implementation support every possible mode. If a mode is not supported, this request should return BAD_MODE.

Offsets for the access routines are returned in the following order:

Set Address
Set Auto Increment
Read Byte
Read Word
Read Byte with Auto Increment
Read Word with Auto Increment
Read Words
Read Words with Auto Increment
Write Byte
Write Word
Write Byte with Auto Increment
Write Word with Auto Increment
Write Words
Write Words with Auto Increment
Compare Byte
Compare Byte with Auto Increment
Compare Words
Compare Words with Auto Increment

Definitions for the arguments passed to the above access routines are binding specific. (See the ***Card Services Specification***.)

See Also **GetSetSSAddr**

5.3.4 GetAdapter [BOTH]

RETCODE = GetAdapter (*Adapter, State, SCRouting*)
ADAPTER *Adapter*;
FLAGS8 *State*;
IRQ *SCRouting*;

The **GetAdapter** service returns the current configuration of the specified adapter.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
AdapterState	O	Current state of the adapter hardware. This parameter can be a combination of the following values:

Value	Meaning
AS_POWERDOWN	If set, adapter hardware is attempting to conserve power. Before using adapter, full power must be restored using the **SetAdapter** service.
	If reset, adapter hardware is fully powered and fully functional.
AS_MAINTAIN	If set, all adapter and socket configuration information is maintained while power consumption is reduced.
	If reset, adapter and socket configuration information must be maintained by the client.
	This value is only valid if the AS_POWERDOWN value is set.

SCRouting	O	Returns status change interrupt routing status. This parameter is an **IRQ** data type. It is a combination of a binary value representing the IRQ level used for routing the status change signal and the following optional bit-masks:

Value	Meaning
IRQ_HIGH	If set, status change interrupt is active-high.
	If reset, status change interrupt is active-low.
IRQ_ENABLE	If set, status change interrupt is enabled. If an unmasked status change event occurs, the adapter generates a hardware interrupt of the specified level.
	If reset, status change interrupts are not generated by the adapter.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid

Comments

Preserving state information may not allow the same level of power reduction as not preserving state information. The ability to reduce power consumption is vendor specific and reduced power settings may not result in any power savings.

All parameters have been designed to map directly to the values required for the **SetAdapter** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use the **SetAdapter** service to modify the configuration without having to create initial values for each parameter.

See Also **InquireAdapter**, **SetAdapter**

5.3.5 GetAdapterCount [BOTH]

RETCODE = GetAdapterCount (*TotalAdapters, Signature*)
COUNT *TotalAdapters*;
SIGNATURE *Signature*;

The **GetAdapterCount** service returns the number of adapters supported by all Socket Services handlers in the host system. It is also used to determine if one or more Socket Services handlers are installed.

Parameter	I/O	Description
TotalAdapters	O	Number of adapters in host environment, if there is a Socket Services handler installed. Must return the total number of adapters in the system, including both 16-bit PC Card-only and CardBus PC Card adapters.
Signature	O	If **RETCODE** is set to SUCCESS and this field is set to the ASCII characters 'SS' on return, there is at least one Socket Services handler installed and *TotalAdapters* is set to the number of adapters in the host environment.

Comments

The client should ensure *Signature* does not contain 'SS' before calling this service. This ensures the client does not use *TotalAdapters* if the routine handling the request does not support Socket Services but still returns SUCCESS.

If a Socket Services handler is not installed, the returned parameters are undefined. Most environments return an undefined value not equal to SUCCESS. However, an environment may use a calling mechanism shared with another, unrelated handler. There is no guarantee the other handler will properly reject an unrecognized Socket Services request. Before accepting the value in *TotalAdapters* as the number of adapters installed, the client must confirm *Signature* contains the ASCII characters 'SS'.

Even if a Socket Services handler is present, there might not be any adapter hardware present. In this case, SUCCESS is returned, *Signature* contains 'SS' and *TotalAdapters* is zero (0). Clients must be prepared for this situation.

Return Codes

SUCCESS	if *Adapter* is valid

See Also **GetSSInfo**

5.3.6 GetBridgeWindow [BOTH]

RETCODE = GetBridgeWindow (*Adapter, Window, Socket, Size, State, Base*)

ADAPTER *Adapter*;
WINDOW *Window*;
SOCKET *Socket*;
SIZE *Size*;
FLAGS8 *State*;
BASE *Base*;

The **GetBridgeWindow** service returns the current configuration of the bridge window specified by the input parameters. If present on the adapter, PC Card bridge windows are required to allow access to devices on PC Cards.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Specifies a bridge window on the adapter.
Socket	O	Physical socket to which the bridge window is currently assigned.
Size	O	Returns the window's current size in bytes.
State	O	Defined as below. Current state of the window hardware. This parameter can be a combination of the following values:

Value	Meaning
WS_IO	If set, this bridge window routes host system I/O accesses to the PC Card socket.
	If reset, this bridge window routes host system memory accesses to the PC Card socket.
WS_ENABLED	If set, the bridge window is enabled and routing host system accesses to a PC Card socket.
	If reset, the bridge window is disabled.
WS_PREFETCH	If set, the bridge window's prefetch hardware is enabled.
	If reset, the bridge window's prefetch hardware is not enabled (or does not exist).
WS_CACHABLE	If set, the bridge window's cache coherency and prefetch hardware are enabled.
	If reset, the bridge window's cache coherency hardware is not enabled (or does not exist).
	Note: All cachable windows are prefetchable.

Base	O	Returns the current base address of the specified bridge window. It is the first address within the host system memory or I/O address space routed to the PC Card socket.

Return Codes

SUCCESS	if *Adapter* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values required for the **SetBridgeWindow** service. This is intended to allow clients of Socket Services to retrieve current configuration

information with this service, make changes and then use the **SetBridgeWindow** service to modify the configuration without having to create initial values for each parameter.

See Also **InquireBridgeWindow**, **SetBridgeWindow**, **InquireWindow**, **GetWindow**, **SetWindow**, **AccessConfigSpace**.

5.3.7 GetEDC [BOTH]

RETCODE = **GetEDC** (*Adapter, EDC, Socket, State, Type*)

ADAPTER	*Adapter*;
EDC	*EDC*;
SOCKET	*Socket*;
FLAGS8	*State*;
FLAGS8	*Type*;

The **GetEDC** service returns the current configuration of the EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.
Socket	O	Returns the physical socket on the adapter that the EDC generator is assigned.
State	O	Returns the current state of the EDC generator. This field may be combination of the following values:

Value	Meaning
EC_UNI	If set, EDC generator is computing in only one direction. EC_WRITE determines whether computation is on read or write accesses.
	If reset, EDC generator is computing on both read and write accesses.
EC_WRITE	If set, EDC generator is computing only on write accesses.
	If reset, EDC generator is computing only on read accesses.
	This value is only valid if EC_UNI is set.

Type	O	Returns type of EDC generated. This parameter may be one of the following values:

Value	Meaning
ET_CHECK8	EDC generated is 8-bit checksum.
ET_SDLC16	EDC generated is 16-bit CRC-SDLC.
ET_SDLC32	EDC generated is 32-bit CRC-SDLC.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

All parameters have been designed to map directly to the values required by the **SetEDC** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use **SetEDC** to modify the configuration without having to create initial values for each parameter.

See Also **InquireEDC**, **SetEDC**, **StartEDC**, **PauseEDC**, **ResumeEDC**, **StopEDC**, **ReadEDC**

5.3.8 GetPage [PC16]

RETCODE = GetPage (*Adapter, Window, Page, State, Offset*)

ADAPTER *Adapter*;
WINDOW *Window*;
PAGE *Page*;
FLAGS8 *State*;
OFFSET *Offset*;

The **GetPage** service returns the current configuration of the page specified by the input parameters. It is only valid for memory windows (WS_IO is reset for the window).

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Specifies a physical window on the adapter.
Page	I	Specifies the page within the *Window*.
State	O	Current state of the *Page* within the *Window*. This parameter can be a combination of the following values:

Value	Meaning
PS_ATTRIBUTE	If set and *Page* is enabled, PC Card attribute memory is mapped into host system memory space.
	If reset and *Page* is enabled, PC Card common memory is mapped into host system memory space.
PS_ENABLED	If set, *Page* is enabled and PC Card is mapped into the host system memory or I/O space.
	If reset, *Page* is disabled.
	Some hardware implementations may not allow individual pages to be disabled, only entire windows. Such implementations always return with PS_ENABLED set unless the entire window is disabled.
PS_WP	If set, *Page* is write-protected by page mapping hardware in socket.
	If reset, *Page* is not write-protected by socket's page-mapping hardware. However, PC Card memory may be write-protected in other ways.

Offset	O	The offset of a PC Card's memory being mapped into host system memory space by this page. The following formula may be used to calculate the system memory address to access the PC Card memory being mapped by the page: Base + (Page * 16 KBytes)

Return Codes

SUCCESS	if *Adapter*, *Page* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_PAGE	if *Page* is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values required for the **SetPage** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use the **SetPage** service to modify the configuration without having to create initial values for each parameter.

All pages in windows which are subdivided into multiple pages are 16 KBytes in size. A window with only a single page may be any size meeting the constraints returned by **InquireWindow**.

To map PC Card memory into system memory requires that both the WS_ENABLED value of the *State* field used by **Get/SetWindow** be set and the PC_ENABLED value of the *State* field used by **Get/SetPage** be set. For windows with WS_PAGED reset, the PS_ENABLED value is ignored by **SetPage**. The window is enabled and disabled by the WS_ENABLED value of **SetWindow**. **GetPage** for windows with WS_PAGED reset reports the value of WS_ENABLED for PS_ENABLED.

For windows with WS_PAGED set, WS_ENABLED acts as a global enable / disable for all pages within the window. Once WS_ENABLED has been set using **SetWindow**, individual pages may be enabled and disabled using **SetPage** and PS_ENABLED.

If WC_WENABLE is reported as set by **InquireWindow**, Socket Services preserves the state of PS_ENABLED for each page in the window whenever WS_ENABLED is changed by **SetWindow**. If WC_ENABLE is reported as reset by **InquireWindow**, the client must use **SetPage** to set the PS_ENABLED state for each page within the window after WS_ENABLED is set with **SetWindow**.

See Also **InquireWindow**, **GetWindow**, **SetWindow**, **SetPage**

5.3.9 GetSetPriorHandler [BOTH]

RETCODE = GetSetPriorHandler (*Adapter, Mode, pHandler*)
ADAPTER *Adapter*;
FLAGS8 *Mode*;
PTR *pHandler*;

The **GetSetPriorHandler** service replaces or obtains the entry point of a prior handler for the Adapter specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Mode	I	Specifies whether the request is to **Get** the prior handler or **Set** a new handler. If *Mode* is zero, the request is to **Get** the prior handler, If *Mode* is one, the request is to **Set** the prior handler.
pHandler	I/O	If *Mode* is **Get** (equal to zero), this parameter is ignored on input and used to return the entry point of the prior handler. If *Mode* is **Set** (equal to one), this parameter contains a pointer to a new prior handler and is used to return the entry point of the old prior handler.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SERVICE	if request is to **Set** a prior handler for a ROM-based handler which is hard-coded to chain to another type of handler
BAD_MODE	if *Mode* is not supported

Comments

If the handler responding to this request is installed in ROM and is the first handler on the Socket Services chain, a request to **Set** the prior handler may be failed.

One reason a **Set** request would fail is the Socket Services it is addressing is in ROM as the first extension of another type of handler which is sharing the call chain. In this case, the vector to the prior handler is probably hard-coded into the ROM and not in RAM prohibiting it from being updated. This should not cause any difficulty to a client wishing to revise the chain, since this Socket Services handler may be bypassed by registering the values returned from a **Get** request to this Socket Services with a replacement Socket Services handler as its prior handler.

Note: The entry point of the prior handler is always returned, even on **Set** requests, if the service succeeds.

WARNING:

*This service should only be used with the first adapter serviced by a Socket Services handler as returned by the **GetSSInfo** service. If a handler services more than one adapter, subsequent requests to the handler for adapters other than the first return the same information and set the same internal data variables.*

WARNING:

To support additional adapters and/or sockets, new Socket Services handlers should be added to the head of the handler chain. Adjusting internal prior handler values should be used only to replace a Socket Services handler with an updated version.

5.3.10 GetSetSSAddr [BOTH]

RETCODE = GetSetSSAddr (*Adapter, Mode, Subfunc, NumAddData, pBuffer*)

ADAPTER *Adapter*;
BYTE *Mode*;
BYTE *Subfunc*;
COUNT *NumAddData*;
PTR *pBuffer*;

The **GetSetSSAddr** service returns code and data area descriptions and provides a way to pass mode-specific data area descriptors to a Socket Services handler.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Mode	I	Specifies the processor mode. This is specific to the type of host platform. See the platform-specific binding for additional detail.
Subfunc	I	Specifies the type of request.
		If *Subfunc* is zero (0), Socket Services returns a description of the code and main data areas in the client-supplied buffer.
		If *Subfunc* is one (1), Socket Services returns a description of any additional data areas in the client-supplied buffer.
		If *Subfunc* is two (2), Socket Services accepts an array of mode-specific pointers to additional data areas in the client-supplied buffer.
		If *Subfunc* is three (3), Socket Services returns a description of the I/O port range or ranges used by the adapter hardware managed by Socket Services in the client supplied buffer.
		If *Subfunc* is four (4), Socket Services returns in the client supplied buffer a description of the main data area and the code area and entry point that utilizes the packet interface
NumAddData	I/O	Number of additional data areas.
		If *Subfunc* is zero (0), Socket Services returns the number of additional data areas in this parameter.
		If *Subfunc* is one (1), the client-supplied buffer returns this number of descriptors for additional data areas.
		If *Subfunc* is two (2), Socket Services accepts this number of mode-specific pointers to additional data areas in the client-supplied buffer.
		If *Subfunc* is three (3), the *NumAddData* field returns the number of I/O address ranges in this parameter.
		If *Subfunc* is four (4), this field is reserved and must be reset to zero (0).
pBuffer	I/O	A pointer to a client-supplied buffer of the appropriate length for the request.
		If *Subfunc* is zero (0), Socket Services returns a description of the code and main data segment in the buffer.
		If *Subfunc* is one (1), Socket Services returns a description of the additional data areas in the client-supplied buffer.
		If *Subfunc* is two (2), the client-supplied buffer contains mode-specific pointers to additional data areas as input to Socket Services.
		If *Subfunc* is three (3), the client supplied buffer contains a list of I/O address ranges that are used to control the sockets for this adapter.
		If *Subfunc* is four (4), Socket Services returns in the buffer a mode-specific entry point that utilizes the packet interface.

Comments

Some Socket Services handlers may require access to other memory regions than their main data area. If this is the case, the value in *NumAddData* reflects the number of unique memory regions the Socket Services handler needs to address besides the main data segment.

A Card Services using an entry point returned by this service is expected to establish the appropriate mode-specific pointers to the code and main data area prior to calling the entry point. When using the entry point returned by this service, the client uses the absolute adapter number within the host environment. For example, if two Socket Services handlers are installed, with the first handler supporting two adapters and the second handler supporting three adapters, the client should use adapter values of zero through one for the first handler and values of two through four for the second handler.

When *Subfunc* is zero (0), the buffer pointed to by *pBuffer* has the following format:

Offset	Size	Description
00H	Double Word	32-bit linear base address of code segment in system memory
04H	Double Word	Limit of code segment
08H	Double Word	Entry point offset
0CH	Double Word	32-bit linear base address of main data segment in system memory
10H	Double Word	Limit of data segment
14H	Double Word	Data area offset

When *Subfunc* is one (1), there are entries in the client-supplied buffer pointed to by *pBuffer* returned for each of the additional data segments. Each entry in the buffer has the following format:

Offset	Size	Description
00H	Double Word	32-bit linear base address of additional data segment
04H	Double Word	Limit of data segment
08H	Double Word	Data area offset

When *Subfunc* is two (2), there are entries in the client-supplied buffer pointed to by *pBuffer* for each additional data area. These entries are mode-specific pointers created by the client for each additional data area. Each entry in the buffer has the following format:

Offset	Size	Description
00H	Double Word	32-bit offset
04H	Double Word	Selector
08H	Double Word	Reserved

When *Subfunc* is three (3), there are entries in the client-supplied buffer pointed to by *pBuffer* for each additional data area. Each entry in the buffer has the following format:

Offset	Size	Description
00H	Double Word	32-bit I/O base address for control ports
04H	Double Word	Number of I/O ports consumed for this entry

When *Subfunc* is four (4), the buffer pointed to by *pBuffer* has the following format:

Offset	Size	Description
00H	Double Word	32-bit linear base address of code segment in system memory
04H	Double Word	Limit of code segment
08H	Double Word	Entry point offset (entry point that utilizes the stack-packet interface)
0CH	Double Word	32-bit linear base address of main data segment in system memory
10H	Double Word	Limit of data segment
14H	Double Word	Data area offset

WARNING:

This service should only be used with the first adapter serviced by a Socket Services handler as returned by the ***GetSSInfo*** *service. If a handler services more than one adapter, subsequent requests to the handler for adapters other than the first return the same information and set the same internal data variables.*

Return Codes

SUCCESS	if *Adapter*, *Mode*, and *Subfunc* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SERVICE	if request is not supported
BAD_MODE	if *Mode* is not supported
BAD_ATTRIBUTES	if number of additional data segments specified when *Subfunc* is one (1) or two (2) does not equal the number of additional data segments returned when *Subfunc* is zero (0)

See Also **GetAccessOffsets**

5.3.11 GetSocket [BOTH]

RETCODE = **GetSocket** (*Adapter, Socket, SCIntMask, Vcontrol, VccLevel, VppLevels, State, CtlInd, IREQRouting, IFType, IFIndex*)

ADAPTER	*Adapter*;
SOCKET	*Socket*;
FLAGS8	*SCIntMask*;
PWRINDEX	*Vcontrol*;
PWRINDEX	*VccLevel*;
PWRINDEX	*VppLevels*;
FLAGS8	*State*;
FLAGS8	*CtlInd*;
IRQ	*IREQRouting*;
FLAGS8	*IFType*;
WORD	*IFIndex;*

The **GetSocket** service returns the current configuration of the socket identified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.
SCIntMask	O	Returns current setting of mask for events that generate a status change interrupt when they occur on the socket. If a value is set the event generates a status change interrupt if the following conditions are met: The event is supported as indicated by the *SCIntCaps* parameter of **InquireSocket** and status change interrupts have been enabled by **SetAdapter**. This parameter is a combination of the SBM_x values defined in **InquireSocket**.
Vcontrol	O	This parameter takes on the following values: **Value** — **Meaning** VCTL_CISREAD — If reset, the **VCC** level and **VPP[2::1]** levels are controlled by the *VccLevel* and *VppLevels* fields. If set, the **VCC** level and **VPP[2::1]** levels are set to the value indicated by the voltage sense signaling from the PC Card. VCTL_OVERRIDE — If reset, the **VCC** level matches the value indicated by the voltage sense signaling from the PC Card. If set, the **VCC** level does not match the value indicated by the voltage sense signaling from the PC Card. Note: The VCTL_CISREAD and VCTL_OVERRIDE bits are mutually exclusive. In addition, Socket Services will reset these bits upon card removal. The following values are mutually exclusive and are only valid when a PC Card is inserted in the socket. They may be read before the socket is powered. They indicate the voltage sense value the PC Card is signaling (**VS[2::1]** signals, see the ***Electrical Specification***) for reading the Card Information Structure (CIS): VCTL_50V — Use 5.0 V to read the CIS. VCTL_33V — Use 3.3 V to read the CIS. VCTL_XXV — Use X.X V to read the CIS. Note: When initially powering a PC Card with SetSocket, if the VCTL_CISREAD value is set, the PC Card is powered to the value indicated by the voltage sense signaling from the card.

VccLevel O Returns current power level of **VCC** signal. This is an index into the array of **PWRENTRY** items returned by **InquireAdapter**. Valid values range from zero to one less than the number of levels returned by **InquireAdapter**.

VppLevels O Returns current power level of **VPP[2::1]** signals. This is two indices into the array of **PWRENTRY** items returned by **InquireAdapter**. Separate values are returned in this parameter for the **VPP1** and **VPP2** signals. Valid values range from zero to one less than the number of levels returned by **InquireAdapter**.

Note: The *VccLevel* and *VppLevels* always return the actual levels currently applied to the card.

State O Returns latched values representing state changes experienced by the socket hardware. Only those values set in the **InquireSocket** *SCRptCaps* parameter will ever be set. Once set, values must be explicitly reset using **SetSocket**.

This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *SCIntCaps* and *SCRptCaps* parameters.

CtlInd O Returns current setting of socket controls and indicators. If a value is set, the corresponding control or indicator is on. If a value is reset, the corresponding control or indicator is off. Values supported by the socket are defined by the *CtlIndCaps* parameter returned by **InquireSocket**.

This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *CtlIndCaps* parameter.

IREQRouting O Returns PC Card **IREQ#** routing status. This parameter is an IRQ data type. It is a combination of a binary value representing the IRQ level used for routing the PC Card **IREQ#** signal and the following optional values:

Value	Meaning
IRQ_HIGH	If set, the PC Card **IREQ#** signal is inverted.
	If reset, the PC Card **IREQ#** signal is routed without inversion.
IRQ_ENABLE	If set, **IREQ#** routing is enabled.
	If reset, **IREQ#** routing is not enabled and interrupts from a PC Card in the socket are ignored.

IFType O Returns the current interface and DMA settings. Only one of the following interface settings is valid at a time: IF_IO, IF_MEMORY, IF_CARDBUS, or IF_CUSTOM. The DREQ and DMA Channel values are only valid if the interface is set to IF_IO and the socket supports DMA as indicated by the IF_DMA value returned by **InquireSocket**.

Value	Meaning
IF_MEMORY	Socket interface is set to Memory-Only. (See the ***Electrical Specification***.)
IF_IO	Socket interface is set to I/O and Memory interface. (See the ***Electrical Specification***.)
IF_CARDBUS	Socket interface is set to CardBus PC Card mode, i.e. the card inserted in the socket is a CardBus PC Card. (See the ***Electrical Specification***.)
IF_CUSTOM	Socket interface is set to a custom interface. The index of the current custom interface is returned in *IFIndex*. (See the ***Electrical Specification*** and see also the ***Metaformat Specification***.)
DREQ	Binary value describing the PC Card signal used for **DREQ#**. If reset to zero, a DMA channel is not currently assigned to this socket.
	If set to one (1), **DREQ#** is assigned to the **SPKR#** pin.
	If set to two (2), **DREQ#** is assigned to the **IOIS16#** pin.
	If set to three (3), **DREQ#** is assigned to the **INPACK#** pin.
DMA Channel	Binary value of the DMA channel currently assigned to this socket. If DREQ is reset to zero, this value is undefined and should be ignored.

IFIndex	O	Returns the current Custom Interface setting when *IFType* is set to IF_CUSTOM. This is an index into the array of *dCustomIF* items returned by **InquireSocket**. Valid values range from zero to one less than the number of interface numbers returned by **InquireSocket**.

Return Codes

SUCCESS	if *Adapter* and *Socket* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SOCKET	if *Socket* is invalid

Comments

All parameters have been designed to map directly to the values required by the **SetSocket** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use **SetSocket** to modify the configuration without having to create initial values for each parameter.

See Also **InquireSocket**, **SetSocket**

5.3.12 GetSSInfo [BOTH]

RETCODE = GetSSInfo (*Adapter, Compliance, NumAdapters, FirstAdapter*)
ADAPTER *Adapter*;
BCD *Compliance*;
COUNT *NumAdapters*;
ADAPTER *FirstAdapter*;

The **GetSSInfo** service returns the compliance level of the Socket Services interface supporting the adapter specified by the input parameters and identifies the adapters serviced by the handler.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system. Each adapter may be handled by a different Socket Services handler. This argument identifies a specific Socket Services handler. If a Socket Services handler supports more than one adapter, the same information is returned for any adapter the handler supports.
Compliance	O	Returns the Socket Services Interface Specification compliance level as a Binary Coded Decimal (BCD) value. If the handler is compliant with Release 12.98 of the *PC Card Socket Services* specification, 1298H is returned.

Publication	***Compliance***
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 5.1 (November 1995)	0510H (5.10)
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)
PCMCIA 1.0 / JEIDA 4.0	0100H (1.00)

NumAdapters	O	Returns the number of adapters supported by this specific Socket Services handler.
FirstAdapter	O	Returns the first adapter number supported by this specific Socket Services handler. The first Socket Services handler installed always returns zero (0) to indicate it supports the first adapter in the system.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid

Example

If a host system had five adapters, two Socket Services handlers and the first handler supported three (3) adapters, this request returns with *FirstAdapter* equal to zero (0) and *NumAdapters* equal to three (3), for *Adapter* values of zero, one or two (0, 1 or 2). If this request was made with *Adapter* set to three or four (3 or 4), it would return with *FirstAdapter* set to three (3) and *NumAdapters* set to two (2).

See Also **GetAdapterCount**

5.3.13 GetStatus [BOTH]

RETCODE = **GetStatus** (*Adapter, Socket, CardState, SocketState, CtlInd, IREQRouting, IFType*)

ADAPTER	*Adapter*;
SOCKET	*Socket*;
FLAGS8	*CardState*;
FLAGS8	*SocketState*;
FLAGS8	*CtlInd*;
IRQ	*IREQRouting*;
FLAGS8	*IFType*;

The **GetStatus** service returns the current status of the card, socket, controls and indicators for the socket identified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.
CardState	O	Returns instantaneous state. This parameter represents the current state of the socket and PC Card, if inserted. It is a combination of the SBM_x values defined in **InquireSocket** for the *SCIntCaps* and *SCRptCaps* parameters. SBM_LOCKED, SBM_EJECT and SBM_INSERT are vendor specific and may not be supported. See **InquireSocket** *SCRptCaps*. For 16-bit PC Cards, SBM_WP is the output of **WP** (pin 33). SBM_BVD1 is the output of **BVD1** (pin 63). SBM_BVD2 the output of **BVD2** (pin 62). SBM_RDYBSY is the output of **READY** (pin 16) and SBM_CD is the AND-ed value of the **CD1#** (pin 36) and **CD2#** (pin 67) outputs. Note that these bits are set when the defined states are true. This is the inverted output of **BVD1**, **BVD2** and the Card Detect signals. If the interface is set to I/O and Memory mode, the meaning of many of these signals change. Values reported are always based on the signal levels at the socket. If the *IFType* is IF_IO, this service does NOT read status from the Pin Replacement Register. This is the responsibility of the client. For CardBus PC Cards, the state of the card is read from the Function Present State register resident on the card. This register shows the current value for such states as SBM_BVD1.
SocketState	O	Returns same latched information as *State* parameter of **GetSocket**. This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *SCIntCaps* and *SCRptCaps* parameters.
CtlInd	O	Returns same information as *CtlInd* parameter of **GetSocket**, the current setting of socket controls and indicators. If a value is set, the corresponding control or indicator is on. If a value is reset, the corresponding control or indicator is off. Values supported by the socket are defined by the *CtlIndCaps* parameter returned by **InquireSocket**. This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *CtlIndCaps* parameter.
IREQRouting	O	Returns same information as *IREQRouting* parameter of **GetSocket**.
IFType	O	Returns the same information as *IFType* parameter of **GetSocket**.

Return Codes

SUCCESS	if *Adapter* and *Socket* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SOCKET	if *Socket* is invalid

WARNING

This service must NOT be invoked during hardware interrupt processing. It is intended to be used by a client during foreground and background processing, but outside of the status change hardware interrupt handler.

See Also **InquireSocket**, **GetSocket**, **SetSocket**

5.3.14 GetVendorInfo[BOTH]

RETCODE = GetVendorInfo (*Adapter, Type, pBuffer, Release*)

ADAPTER *Adapter*;
BYTE *Type*;
PTR *pBuffer*;
BCD *Release*;

The **GetVendorInfo** service returns information about the vendor implementing Socket Services for the adapter specified in the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Type	I	Specifies the type of vendor information to return in the client-supplied buffer. The only *Type* currently defined is zero (0) which is an ASCIIZ string describing the implementer.

pBuffer	I	If *Type* is zero (0), this parameter points to a client-supplied buffer to be filled with an ASCIIZ string describing the implementer. The buffer has the following form:
		`typedef struct tagVISTRUCT {`
		`  WORD wBufferLength = (BUF_SIZE - 4);`
		`  WORD wDataLength;`
		`  char szImplementor[BUF_SIZE - 4];`
		`} VISTRUCT;`
		The **wBufferLength** field is set by the client to the length of the **VISTRUCT** structure provided less the size of the first two fields (4 bytes). The **wDataLength** field is set by Socket Services to the size of the information it has to return. Only the information that fits in the buffer is copied. If the **wDataLength** is greater than **wBufferLength**, the information is truncated.

Release	O	Vendor's release number in BCD format. Each time a vendor releases a new version of their Socket Services handler, they should change the value returned. The initial *Release* should use the value 0100H to represent Release 1.00 of a vendor's Socket Services handler. A subsequent release of an updated version compliant with the same level of the Socket Services Interface Specification should change this value according to the vendor's change control procedures. The first release of an Socket Services handler compliant with a new specification should again use 0100H to indicate this is the vendor's first release compliant with the new Socket Services specification. Each Socket Services released by a vendor must be uniquely identified by the combination of the compliance level returned by the *Compliance* parameter of **GetSSInfo** and this parameter.

Return Codes

SUCCESS	if *Adapter* and *Type* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SERVICE	if *Type* is invalid

5.3.15 GetWindow [PC16]

RETCODE = GetWindow (*Adapter, Window, Socket, Size, State, Speed, Base*)

ADAPTER *Adapter*;
WINDOW *Window*;
SOCKET *Socket*;
SIZE *Size*;
FLAGS8 *State*;
SPEED *Speed*;
BASE *Base*;

The **GetWindow** service returns the current configuration of the window specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	*Window* number. Specifies a physical window on the adapter.
Socket	O	Returns the physical socket the *Window* is currently assigned. Socket numbers range from zero to fifteen using bits 0 to 3. The rest of the bits in this field are binding specific.
Size	O	Returns the window's current size. If *Size* is equal to zero (0), the window is the maximum size that may be represented by the data type used for this parameter plus one. For example, if the data type used for *Size* is a word and it is expressed in units of a byte, a value of zero represents a window size of 65,536 bytes.
State	O	Defined as below. Current state of the window hardware. This parameter can be a combination of the following values:

Value	Meaning
WS_IO	If set, window maps registers on a 16-bit PC Card into the host system's I/O address space.
	If reset, window maps memory address space on a 16-bit PC Card into the host system's memory address space.
WS_ENABLED	If set, window is enabled and mapping a card's address space into the host system memory or I/O address space.
	If reset, window is disabled.
WS_16BIT	If set, window is programmed for a 16-bit data bus width.
	If reset, window is programmed for an 8-bit data bus width.
WS_PAGED	If set, window is subdivided into multiple 16 KByte pages whose card offset addresses may be set individually using **SetPage**.
	If reset, window is a single page.
	This value is only valid for memory windows (WS_IO reset).
WS_EISA	If set, window is using EISA I/O mapping.
	If reset, window is using ISA I/O mapping.
	This value is only valid for I/O windows (WS_IO set).
WS_CENABLE	If set, accesses to I/O ports in EISA common I/O areas generate card enables.
	If reset, accesses to I/O ports in EISA common I/O areas are ignored.
	This value is only valid for I/O windows (WS_IO set) that have WS_EISA set.

Speed	O	This parameter is the actual access speed being used by the window. It uses the format of the Device Speed Code and Extended Device Speed Codes of the Device Information Tuple. (See the ***Metaformat Specification***.) The Device Speed Code Values are used when what would be the mantissa of an Extended Device Speed Code is reset to zero (0). If the mantissa is non-zero, supported device speeds are coded according to the Extended Device Speed Code. This parameter may not match the value specified by a successful **SetWindow** request. If Socket Services does not support the speed requested, it uses the next slowest speed it supports. For Socket Services, Bit 7 of *Speed* is reserved and is reset to zero (0). This parameter is not used and should be ignored for I/O windows (WS_IO set).
Base	O	Returns the current base address of the specified window. It is the first address within the host system memory or I/O address space to which the window responds.

Return Codes

SUCCESS	if *Adapter* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values required for the **SetWindow** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use the **SetWindow** service to modify the configuration without having to create initial values for each parameter.

For memory mapping windows, the area of the PC Card memory array mapped into the host system memory space may be managed by **GetPage** and **SetPage** requests.

To map 16-bit PC Card memory into system memory requires that both the WS_ENABLED value of the *State* field used by **Get/SetWindow** be set and the PS_ENABLED value of the *State* field used by **Get/SetPage** be set. For windows with WS_PAGED reset, the PS_ENABLED value is ignored by **SetPage**. The window is enabled and disabled by the WS_ENABLED value of **SetWindow**. **GetPage** for windows with WS_PAGED reset reports the value of WS_ENABLED for PS_ENABLED.

For windows with WS_PAGED set, WS_ENABLED acts as a global enable / disable for all pages within the window. Once WS_ENABLED has been set using **SetWindow**, individual pages may be enabled and disabled using **SetPage** and PS_ENABLED.

If WC_WENABLE is reported as set by **InquireWindow**, Socket Services preserves the state of PS_ENABLED for each page in the window whenever WS_ENABLED is changed by **SetWindow**. If WC_ENABLE is reported as reset by **InquireWindow**, the client must use **SetPage** to set the PS_ENABLED state for each page within the window after WS_ENABLED is set with **SetWindow**.

See Also **InquireWindow**, **SetWindow**, **GetPage**, **SetPage**, **InquireBridgeWindow**, **GetBridgeWindow**, **SetBridgeWindow**

5.3.16 InquireAdapter [BOTH]

RETCODE = **InquireAdapter** (*Adapter, pBuffer, NumSockets, NumWindows, NumEDCs, NumBridgeWindows*)

ADAPTER *Adapter*;
PTR *pBuffer*;
COUNT *NumSockets*;
COUNT *NumWindows*;
COUNT *NumEDCs*;
COUNT *NumBridgeWindows*;

The **InquireAdapter** service returns information about the capabilities of the adapter specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.

pBuffer	I	Points to a client-supplied buffer to be filled with information about the adapter. The buffer has the following form:
		`typedef struct tagAISTRUCT {`
		`  WORD wBufferLength;`
		`  WORD wDataLength;`
		`  ACHARTBL CharTable;`
		`  WORD wNumPwrEntries = NUM_ENTRIES;`
		`  PWRENTRY PwrEntry[NUM_ENTRIES];`
		`} AISTRUCT;`
		The *wBufferLength* field is set by the client to the size in bytes of **AISTRUCT** less the size of the first two fields (4 bytes). The *wDataLength* field is set by Socket Services to the size of the information it has to return. Only the information that fits in the buffer is copied. If the *wDataLength* is greater than *wBufferLength*, the information is truncated.
		The **ACHARTBL** structure is defined on page 46.
		A **PWRENTRY** is a structure which has two members. One member is a binary value representing a DC voltage level in tenth of a volt increments (25.5 V DC maximum). The other member indicates which power signals may be set to the specified voltage level. It may be set to a combination of the following: **VCC**, **VPP1**, and/or **VPP2**.
		A **PWRENTRY** is a structure which has two members. One member is a binary value representing a DC voltage level in tenth of a volt increments (25.5 V DC maximum). The other member indicates which power signals may be set to the specified voltage level. It may be set to a combination of the following: **VCC**, **VPP1**, and/or **VPP2**.
		The **PWRENTRY** structure is defined on page 47.

NumSockets	O	Returns the number of sockets provided by the adapter.
NumWindows	O	Returns the number of 16-bit PC Card windows provided by the adapter.
NumEDCs	O	Returns the number of Error Detection Code (EDC) generators provided by the adapter.
NumBridgeWindows	O	Returns the number of bridge windows provided by the adapter.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid

Comments

By convention, all sockets on an adapter use the same PWRENTRY array. There is one **PWRENTRY** for each supported voltage.

The **PWRENTRY** only indicates it is possible to set one or more of the power pins to that power level. The **PWRENTRY** does not indicate acceptable power combinations for the power pins. The example below indicates **VCC**, **VPP1** and **VPP2** can be set to No Connect and that **VPP1** and **VPP2** can be set to 12 V. This table does not define any relationships between **VPP1**, **VPP2**, and **VCC**; for example, implementations may fail requests to set **VCC** to 0 V and either **VPP1** or **VPP2** to 12 V. . It is up to the Socket Services client to determine if a particular combination of power levels is valid for the PC Card in the socket.

Example

```
AISTRUCT AdapterInfo = {
      18,      //Size of client-supplied buffer is 18 bytes
      18,      //Size of data returned is 18 bytes
      {0,      //Indicators, power and data bus width
         //controlled at the socket
      0, //No cache support on adapter
      0xDEB8,  //Status changes may be routed to IRQ levels
         //    3, 4, 5, 7, 9, 10, 11, 12, 14, and 15
         //    as an active high signal
      0},      //Status changes are not available on any
         //    level as an active low signal
      3, //Number of PWRENTRY elements
      ((VCC | VPP1 | VPP2) << 8) | 0,  // Vcc, Vpp1 and Vpp2 - No connect
      ((VCC | VPP1 | VPP2) << 8) | 50, // Vcc, Vpp1 and Vpp2 - 5.0 VDC
      ((VPP1 | VPP2) << 8) | 120       // Vpp1 and Vpp2   - 12.0 VDC
};
```

See Also **GetAdapter**, **SetAdapter**

Adapter Characteristics Structure

```
typedef struct tagACHARTBL {
      FLAGS8   AdpCaps;
      BYTE     CacheLineSize;
      FLAGS32  ActiveHigh;
      FLAGS32  ActiveLow;
} ACHARTBL;
```

Member	Description
AdpCaps	Flags indicating whether certain characteristics are controlled at an adapter level or at a socket level. If set, the characteristic is controlled at the adapter level. This member can be a combination of the following values:

Value	Meaning
AC_IND	Indicators - If AC_IND is set, indicators for write-protect, card lock, battery status, busy status and XIP status are shared for all sockets on the adapter. If AC_IND is reset, there are individual indicators for each socket on the adapter.
AC_PWR	Power Level - If AC_PWR is set, even though the interface provides for separate power level controls for each socket using the **SetSocket** service, the adapter requires that all sockets be set to the same value. Socket Services is responsible for resolving conflicts between settings for individual sockets. When the AC_PWR flag is set, setting **VPP[2::1]** to 12 V results in 12 V being applied to all of the sockets on the adapter. Socket Services does not remove 12 V from the **VPP[2::1]** lines until all sockets set **VPP[2::1]** back to the **VCC** level. If AC_PWR is reset, power levels may be individually set for each socket on the adapter.
AC_DBW	Data Bus Width - If AC_DBW is set, all windows on the adapter must use the same data bus width. If AC_DBW is reset, the data bus width is set individually for each window on the adapter.
AC_CARDBUS	CardBus PC Card capable. If set, all sockets are CardBus PC Card-capable. If reset, then all sockets are not CardBus PC Card-capable.

CacheLineSize	Host system cache line size in units of 32-bit words. CardBus PC Cards participating in caching protocol need this information to know when to retry burst accesses at cache line boundaries. If bridge windows on the adapter do not support caching or there are no bridge windows on the adapter, this field is reset to zero. This field is also reset to zero if the adapter does not support CardBus PC Cards.
ActiveHigh	Bit-map of IRQ levels the Status Change interrupt may be routed with an active high state when an unmasked event occurs.
ActiveLow	Bit-map of IRQ levels the Status Change interrupt may be routed with an active low state when an unmasked event occurs.

Power Entry Structure

```
typedef struct tagPWRENTRY {
      PWRINDEX PowerLevel;
      FLAGS8   ValidSignals;
} PWRENTRY;
```

Member	Description
PowerLevel	DC voltage level in tenth of a volt increments. The power level ranges from zero (meaning No Connect) to 25.5 V DC in tenth of a volt increments.
ValidSignals	Flags indicating whether voltage is valid for specific signals. This member can be a combination of the following values:

Value	Meaning
VCC	Voltage level is valid for **VCC** signal.
VPP1	Voltage level is valid for **VPP1** signal.
VPP2	Voltage level is valid for **VPP2** signal.

5.3.17 InquireBridgeWindow [BOTH]

RETCODE = InquireBridgeWindow (*Adapter, Window, pBuffer, WndCaps, Sockets*)

ADAPTER *Adapter*;
WINDOW *Window*;
PTR *pBuffer*;
FLAGS8 *WndCaps*;
SKTBITS *Sockets*;

The **InquireBridgeWindow** service returns information about the capabilities of the bridge window specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Specifies a bridge window on the adapter.

pBuffer	I	Points to a client-supplied buffer to be filled with information about the bridge window. The buffer has the following form:
		`typedef struct tagBWISTRUCT {`
		`  WORD wBufferLength;`
		`  WORD wDataLength;`
		`  BWINTBL WinTable[NUM_TYPES];`
		`} BWISTRUCT;`
		The *wBufferLength* field is set by the client to the size in bytes of **BWISTRUCT** less the size of the first two fields (4 bytes). The *wDataLength* field is set by Socket Services to the size of the information it has to return. Only the information that fits in the buffer is copied. If the *wDataLength* is greater than *wBufferLength*, the information is truncated.
		A bridge window may support either I/O or memory accesses. Each window type has associated characteristics described in tables returned in the client-supplied buffer.
		Bridge window characteristics vary if the hardware is used as a memory or as an I/O window. For that reason, this service provides two tables of information. The **BMEMWINTBL** structure is defined on page 50. The **BIOWINTBL** structure is defined on page 52.
		If a bridge window supports both memory and I/O access, both characteristics tables are copied to the client-supplied buffer. When a bridge window supports both types of access, the memory window characteristics table is first in the buffer, followed by the I/O window characteristics table. If only one type of access is supported, only the appropriate characteristics table is copied into the buffer by Socket Services.

WndCaps	O	This parameter indicates the capability of the specified window. It can be a combination of the following values: **Value** — **Meaning** WC_IO — If set, bridge window may be used to route host system I/O accesses to a PC Card socket. WC_MEMORY — If set, bridge window may be used to route host system memory accesses to a PC Card socket.
Sockets	O	Depending on the hardware implementation, bridge windows may be dedicated to a particular socket or may allow assignment to a given socket on an adapter. If a bridge window may be assigned to a socket on the adapter, the corresponding bit in this parameter is set. If a socket does not exist on an adapter its corresponding bit is reset. The first socket on the adapter is represented by the least significant bit of this parameter. Note: The size of this field constrains the number of sockets that may be supported by an adapter.

Return Codes:

SUCCESS	if *Adapter* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_WINDOW	if *Window* is invalid

See Also **GetBridgeWindow**, **SetBridgeWindow**, **InquireWindow**, **GetWindow**, **SetWindow**, **AccessConfigSpace**.

Bridge Memory Window Characteristics Table

```
typedef struct tagBMEMWINTBL {
      FLAGS16  MemWndCaps;
      BASE     FirstByte;
      BASE     LastByte;
      SIZE     MinSize;
      SIZE     MaxSize;
      SIZE     ReqGran;
      SIZE     ReqBase;
} BMEMWINTBL;
```

Member	Description	
MemWndCaps	Flags indicating memory bridge window characteristics. This member can be a combination of the following values.	
	Value	**Meaning**
	WC_BASE	If set, the base address of the bridge window is programmable within the range specified by the *FirstByte* and *LastByte* members. If reset, the base address of the bridge window is fixed in system memory space at the address specified in the *FirstByte* member. When reset, the *LastByte* member is undefined.
	WC_SIZE	If set, the bridge window size is programmable within the range specified by the *MinSize* and *MaxSize* members. If reset, the bridge window size is fixed to the size indicated by the *MinSize* member. When reset, both the *MinSize* and *MaxSize* members should be the same value.
	WC_WENABLE	If set, the window may be disabled and enabled without reprogramming its characteristics. If reset, the client must preserve window state information before disabling the window.
	WC_BALIGN	If set, the bridge window base address must be programmed to align with a multiple of the bridge window size. For example, a bridge window 16 MBytes in size needs to start on a 16 MByte boundary in the host system memory address space. If reset, the bridge window base address may be programmed anywhere in the bridge window's valid range, subject to any constraint specified by *ReqBase*.
	WC_POW2	If set, a bridge window with WC_SIZE also set must be sized between the *MinSize* and *MaxSize* members as a power of two of the *ReqGran* member. If reset, a bridge window with WC_SIZE set may be any multiple of the *ReqGran* member between the *MinSize* and *MaxSize* members.
	WC_FETCHABLE	If set, this bridge window supports prefetching CardBus PC Card memory. If reset, this bridge window does not support prefetching CardBus PC Card memory.
	WC_CACHABLE	If set, this bridge window supports caching CardBus PC Card memory. If reset, this bridge windows does not support caching CardBus PC Card memory.

FirstByte	First byte addressable in host system memory address space by bridge window. If bridge window base is not programmable, this is the fixed base address of the bridge window.
LastByte	Last byte addressable in host system memory address space by bridge window. The last byte of the bridge window (base address programmed plus bridge window size minus one) may not exceed this value. If bridge window base is not programmable, this member is undefined.
MinSize	The minimum bridge window size. When a bridge window size is programmed with **SetBridgeWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements.
MaxSize	The maximum bridge window size. When bridge window size is programmed with **SetBridgeWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements. The bridge window size may be further limited by the base address of the bridge window. The base address plus the bridge window size minus one must not exceed the *LastByte* member for bridge windows with programmable sizes. If *MaxSize* is zero, bridge window size is the largest value that may be represented by the **SIZE** data type plus one.
ReqGran	This member describes the required units for expressing bridge window size due to hardware constraints. If the bridge window size is fixed (WC_SIZE is reset), this member will be the same as the *MinSize* and *MaxSize* members.
ReqBase	If WC_BALIGN is reset, this member describes any alignment boundary requirement for programming the bridge window's base address with **SetBridgeWindow**. If WC_BALIGN is set, this field is undefined.

Comments

Memory bridge windows are used to route host system memory accesses to a PC Card. Not all adapters have bridge windows. For those that do, both a bridge window and the hardware used to map PC Card memory address space into the host system must be enabled at overlapping addresses. For 16-bit PC Cards, a mapping window on the adapter and one or more pages within the window must be enabled. For CardBus PC Cards, a Base Address Register on the card must be programmed. A single bridge window assigned to a socket may be used with multiple mapping windows or Base Address Registers.

For CardBus PC Cards, a given bridge window may be capable of both prefetchable and cacheable memory accesses. However, only one of these capabilities may be enabled at a time. The characteristics of the PC Card memory accessed by programming a Base Address Register on the card must match the bridge window’s characteristics.

Bridge I/O Window Characteristics Table

```
typedef struct tagBIOWINTBL {
      FLAGS16  IOWndCaps;
      BASE     FirstByte;
      BASE     LastByte;
      SIZE     MinSize;
      SIZE     MaxSize;
      SIZE     ReqGran;
} BIOWINTBL;
```

Member	Description
IOWndCaps	Flags indicating I/O bridge window characteristics. This member can be a combination of the following values:

Value	Meaning
WC_BASE	If set, the base address of the bridge window is programmable within the range specified by the *FirstByte* and *LastByte* members. If reset, the base address of the bridge window is fixed in system I/O address space at the address specified in the *FirstByte* member. When WC_BASE is reset, the *LastByte* member is undefined.
WC_SIZE	If set, the bridge window size is programmable within the range specified by the *MinSize* and *MaxSize* members. If reset, the bridge window size is fixed to the size indicated by the *MinSize* member. When WC_SIZE is reset, both the *MinSize* and *MaxSize* members shall be the same value.
WC_WENABLE	If set, the bridge window may be disabled and enabled without reprogramming its characteristics. If reset, the client must preserve bridge window state information before disabling the window.
WC_POW2	If set, a bridge window with WC_SIZE set must be sized between the *MinSize* and *MaxSize* members as a power of two of the *ReqGran* member. If reset, a bridge window with WC_SIZE set may be any multiple of the *ReqGran* member between the *MinSize* and *MaxSize* members.

FirstByte	First byte addressable in host system I/O address space by the bridge window. If the bridge window base is not programmable, this is the fixed base address of the bridge window.
LastByte	Last byte addressable in host system I/O address space by the bridge window. The last byte of the bridge window (base address programmed plus window size minus one) may not exceed this value. If the bridge window base is not programmable, this member is undefined. If *LastByte* is expressed in units other than bytes, any address bits of lesser significance not directly expressed are assumed to be set to one (1). For example, if *LastByte* is expressed in 4 KByte units, a value of A3H indicates the last addressable byte within the window is at location A3FFFH in the host system's I/O address space.

MinSize	The minimum bridge window size. When bridge window size is programmed with **SetBridgeWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements.
MaxSize	The maximum bridge window size. When bridge window size is programmed with **SetBridgeWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements. The bridge window size may be further limited by the base address of the bridge window. The base address plus the bridge window size minus one must not exceed the *LastByte* member for bridge windows with programmable sizes. If *MaxSize* is zero, bridge window size is the largest value that may be represented by the **SIZE** data type plus one.
ReqGran	This member describes the required units for expressing bridge window size due to hardware constraints. If the bridge window size is fixed (WC_SIZE is reset), this member will be the same as the *MinSize* and *MaxSize* members.

I/O bridge windows are used to provide access to the host system I/O address space for PC Card windows. Not all adapters have bridge windows. For those that do, both a bridge window and the hardware used to map PC Card I/O address space into the host system must be enabled at overlapping addresses.

5.3.18 InquireEDC [BOTH]

RETCODE = InquireEDC (*Adapter, EDC, Sockets, Caps, Types*)

ADAPTER *Adapter*;
EDC *EDC*;
SKTBITS *Sockets*;
FLAGS8 *Caps*;
FLAGS8 *Types*;

The **InquireEDC** service returns the capabilities of the EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.
Sockets	O	A bit-map of the sockets the EDC generator may be assigned.
Caps	O	Returns the capabilities of the EDC generator. This field may be combination of the following values:

Value	Meaning
EC_UNI	If set, EDC generator supports unidirectional code generation.
	If reset, EDC generator does not support unidirectional code generation.
EC_BI	If set, EDC generator supports bi-directional code generation.
	If reset, EDC generator does not support bi-directional code generation.
EC_REGISTER	If set, EDC generation is supported through register-based access.
	If reset, EDC generation is not supported through register-based access.
EC_MEMORY	If set, EDC generation is supported during window access.
	If reset, EDC generation is not supported during window access.
EC_PAUSABLE	If set, EDC generation can be paused.
	If reset, EDC generation cannot be paused.
	This value is set if the EDC generator may be paused during computation. This allows algorithms which require multiple accesses to a single location on a card from computing an erroneous EDC value.
	If this value is not set, the **PauseEDC** and **ResumeEDC** services are not available.

Types	O	Returns types of EDC generation supported. This parameter may be a combination of the following values:

Value	Meaning
ET_CHECK8	If set, EDC generator supports 8-bit checksum code generation.
	If reset, EDC generator does not support 8-bit checksum code generation.
ET_SDLC16	If set, EDC generator supports 16-bit CRC-SDLC code generation.
	If reset, EDC generator does not support 16-bit CRC-SDLC code generation.
ET_SDLC32	If set, EDC generator supports 32-bit CRC-SDLC code generation.
	If reset, EDC generator does not support 32-bit CRC-SDLC code generation.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

A hardware implementation may or may not provide EDC generation. This service describes the capability of a particular EDC generator. EDC generators may be shared between sockets. Higher-level software must arbitrate the use of EDC generators.

If EDC generation is available, **InquireAdapter** returns the number of EDC generators available for all the sockets supported by the adapter. The capabilities of each generator can be enumerated by calling this service for each generator.

Socket Services supports two types of EDC generation: checksums for 8-bit transfers and CRC-SDLC calculations for 16-bit and 32-bit transfers. EDC generation may be produced by read or write accesses. Special programming algorithms which require a combination of reads and writes must be aware of how EDC generation is performed to avoid erroneous computations. Bi-directional EDC generation may not be usable with Flash programming algorithms since these algorithms typically require a combination of reads and writes.

EDC generation may not be available with memory-mapped implementations. EDC generators must be configured before use with the **SetEDC** service.

See Also **GetEDC**, **SetEDC**, **StartEDC**, **PauseEDC**, **ResumeEDC**, **StopEDC**, **ReadEDC**

5.3.19 InquireSocket [BOTH]

RETCODE = InquireSocket (*Adapter, Socket, pBuffer, SCIntCaps, SCRptCaps, CtlIndCaps*)

ADAPTER *Adapter*;
SOCKET *Socket*;
PTR *pBuffer*;
FLAGS8 *SCIntCaps*;
FLAGS8 *SCRptCaps*;
FLAGS8 *CtlIndCaps*;

The **InquireSocket** service returns information about the capabilities of the socket specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.

pBuffer	I	Points to a client-supplied buffer to be filled with information about the socket. The buffer has the following form:
		`typedef struct tagSISTRUCT {`
		`  WORD wBufferLength;`
		`  WORD wDataLength;`
		`  SCHARTBL CharTable;`
		`} SISTRUCT;`
		The *wBufferLength* field is set by the client to the size in bytes of **SISTRUCT** less the size of the first two fields (4 bytes). The *wDataLength* field is set by Socket Services to the size of the information it has to return. Only the information that fits in the buffer is copied. If the *wDataLength* is greater than *wBufferLength*, the information is truncated.
		The **SCHARTBL** structure is defined below.

SCIntCaps	O	Returns a bit-map of events which can trigger a Status Change interrupt. If an event can trigger a status change interrupt, its value in this parameter is set. In order for the event to trigger a status change event on a socket, the corresponding value in the *SCIntMask* parameter of **SetSocket** must be set and status change interrupts must be enabled. For 16-bit PC Cards the following values are implemented as signals. For CardBus PC Cards several values are read from the Function Present State register on the card, rather than being implemented as individual signals. This parameter is a combination of the values described below:

Value	Meaning
SBM_WP	PC Card **WP** (write-protect).
SBM_LOCKED	Externally generated indicating the state of a mechanical or electrical card lock mechanism. Not the same as SBM_LOCK which is used to control a card lock.
SBM_EJECT	Externally generated indicating a request to eject a PC Card from the socket has been made.
SBM_INSERT	Externally generated indicating a request to insert a PC Card into the socket has been made.
SBM_BVD1	PC Card **BVD1**. When set, this indicates the battery is no longer serviceable.
SBM_BVD2	PC Card **BVD2**. When set, this indicates the battery is weak.
SBM_RDYBSY	PC Card **READY**.
SBM_CD	**CD1#** and **CD2#** (16-bit PC Card) or **CCD1#** and **CCD2#** (CardBus PC Card).

SCRptCaps	O	Returns Status Change events that the socket is capable of reporting. This parameter is not the same as *SCIntCaps*. Some events may be reportable by **GetStatus**, but not able to generate a status change interrupt as indicated by *SCIntCaps*. If an event is not reportable by **GetStatus**, its value in this parameter is reset. In this case, corresponding values in the **GetStatus** *CardStatus* parameter are undefined. This parameter is a combination of the SBM_x values described under the *SCIntCaps* parameter.
CtlIndCaps	O	Returns control and indicator capabilities of the socket. If a value is set, the control or indicator is supported. If a value is reset, the control or indicator is not supported. This parameter may be a combination of the following values:

Value	Meaning
SBM_WP	Indicator for PC Card **WP** (write-protect) state.
SBM_LOCKED	Indicator for externally generated event indicating the state of a mechanical or electrical card lock mechanism
SBM_EJECT	Control for motor to eject a PC Card from the socket.
SBM_INSERT	Control for motor to insert a PC Card into the socket.
SBM_LOCK	Control for card lock. Not the same as SBM_LOCKED which reflects the state of an externally generated card lock event.
SBM_BATT	Indicator for **BVD1** and **BVD2** state.
SBM_BUSY	Indicator for showing card is in-use.
SBM_XIP	Indicator for eXecute-In-Place application in progress.

Return Codes

SUCCESS	if *Adapter* and *Socket* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_SOCKET	if *Socket* is invalid

Example

```
SISTRUCT SocketInfo = {
      20,      // Size of client-supplied buffer is 20 bytes
      20,      // Size of data returned is 20 bytes
      {IF_MEMORY | IF_IO,  // Socket supports Memory-Only and
                           // I/O and Memory interfaces
      0xDEB8,  // PC Card IREQ# signal may be routed to IRQ levels
               //    3, 4, 5, 7, 9, 10, 11, 12, 14, and 15
               //    as an active high signal
      0,       // PC Card IREQ# routing not available on any
               //    level as an active low signal
      2,       // Number of custom interfaces supported
      0x0141,  // Custom Interface Number dCustomIF[0], index 0
      0x0241}, // Custom Interface Number dCustomIF[1], index 1
};
```

See Also **GetSocket**, **SetSocket**

Socket Characteristics Structure

```
typedef struct tagSCHARTBL {
      FLAGS16  SktCaps;
      FLAGS32  ActiveHigh;
      FLAGS32  ActiveLow;
      FLAGS16  DMAChannels;
      WORD     wNumCustomIF = NUM_ENTRIES;
      DWORD    dCustomIF[NUM_ENTRIES];
} SCHARTBL;
```

Member	Description
SktCaps	Flags indicating socket characteristics. If set, the characteristic is supported. This member can be a combination of the following values:

Value	Meaning
IF_MEMORY	Socket supports Memory-Only interface. (See the ***Electrical Specification***.)
IF_IO	Socket supports I/O and Memory interface. (See the ***Electrical Specification***.)
IF_CB	Socket supports CardBus PC Card interface. (See the ***Electrical Specification***.)
IF_33VCC	Socket supports 3.3 V Interface.
IF_XXVCC	Socket supports X.X V Interface.
IF_VSKEY	Socket supports Low Voltage Key.
IF_DMA	Socket supports 16-bit PC Card DMA transfers. (See the ***Electrical Specification*** and the ***Card Services Specification***.)

ActiveHigh	Bit-map of IRQ levels available for routing an inverted PC Card **IREQ#** signal when an unmasked event occurs.
ActiveLow	Bit-map of IRQ levels available for routing the normal PC Card **IREQ#** signal when an unmasked event occurs. It is assumed that PC Card **IREQ#** signals may be shared in a host system.
DMAChannels	Bit-map of DMA channels supported by this socket. Bit 0 through 15 correspond to DMA channel 0 through 15. If a bit is set to one, the corresponding DMA channel is supported by the socket. The DMA width supported by any DMA channel Is host system specific and beyond the scope of Socket Services. If a socket does not support DMA operations, this field may be omitted.
wNumCustomIF	The number of custom interfaces supported by this socket. If this number is non-zero the socket supports custom interfaces in addition to the interfaces indicated in the *SktCaps* field.
dCustomIF	Array of custom interface ID numbers supported by this socket. (See the ***Electrical Specification*** and see also the ***Metaformat Specification***.)

5.3.20 InquireWindow [PC16]

RETCODE = InquireWindow (*Adapter, Window, pBuffer, WndCaps, Sockets*)

ADAPTER *Adapter*;
WINDOW *Window*;
PTR *pBuffer*;
FLAGS8 *WndCaps*;
SKTBITS *Sockets*;

The **InquireWindow** service returns information about the capabilities of the window specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	*Window* number. Specifies a physical window on the adapter

pBuffer	I	Points to a client-supplied buffer to be filled with information about the window. The buffer has the following form:
		`typedef struct tagWISTRUCT {`
		`  WORD wBufferLength;`
		`  WORD wDataLength;`
		`  WINTBL WinTable[NUM_TYPES];`
		`} WISTRUCT;`
		The *wBufferLength* field is set by the client to the size in bytes of **WISTRUCT** less the size of the first two fields (4 bytes). The *wDataLength* field is set by Socket Services to the size of the information it has to return. Only the information that fits in the buffer is copied. If the *wDataLength* is greater than *wBufferLength*, the information is truncated.
		A window may support two types of mapping: memory or I/O. Each window type has associated characteristics described in tables returned in the client-supplied buffer.
		Window characteristics vary if the hardware is used as a memory or as an I/O window. For that reason, this service may provide multiple tables of information. The **MEMWINTBL** structure is defined on page 61. The **IOWINTBL** structure is defined on page 65.
		If a window supports both memory and I/O mapping, both characteristics tables are copied to the client-supplied buffer. When a window supports both types of mapping, the memory window characteristics table is first in the buffer, followed by the I/O window characteristics table. If only one type of mapping is supported, only the appropriate characteristics table is copied into the buffer by Socket Services.
		EISA I/O and Memory windows may be selected, but the supported I/O map is not programmable. Card enables are asserted based on the pre-defined address line settings returned in the I/O window characteristics structure member *EISASlot*.

WndCaps	O	This parameter indicates the capability of the specified window. It can be a combination of the following values:

Value	Meaning
WC_COMMON	If set, window may be used to map the common memory plane of a 16-bit PC Card into the host system memory address space.
WC_ATTRIBUTE	If set, window may be used to map the attribute memory plane of a 16-bit PC Card into the host system memory address space.
WC_IO	If set, window may be used to map I/O ports on a 16-bit PC Card into the host system I/O address space.
WC_WAIT	If set, window supports the use of the **WAIT#** signal from a 16-bit PC Card to generate additional wait states.

Sockets	O	Depending on the hardware implementation, windows may be dedicated to a particular socket or may allow assignment to one or more sockets on an adapter. If a window may be assigned to a socket, the corresponding bit in this parameter is set. If a socket does not exist on an adapter its corresponding bit is reset. The first socket on the adapter is represented by the least significant bit of this parameter. Note: The size of this field constrains the number of sockets that may be supported by an adapter.

Return Codes

SUCCESS	if *Adapter* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_WINDOW	if *Window* is invalid

See Also **GetWindow**, **SetWindow**, **InquireBridgeWindow**, **GetBridgeWindow**, **SetBridgeWindow**

Memory Window Characteristics Table

```
typedef struct tagMEMWINTBL {
      FLAGS16  MemWndCaps;
      BASE     FirstByte;
      BASE     LastByte;
      SIZE     MinSize;
      SIZE     MaxSize;
      SIZE     ReqGran;
      SIZE     ReqBase;
      SIZE     ReqOffset;
      SPEED    Slowest;
      SPEED    Fastest;
} MEMWINTBL;
```

Member	Description	
MemWndCaps	Flags indicating memory window characteristics. This member can be a combination of the following values:	
	Value	**Meaning**
	WC_BASE	If set, the base address of the window is programmable within the range specified by the *FirstByte* and *LastByte* members. If reset, the base address of the window is fixed in system memory address space at the address specified in the *FirstByte* member. When reset, the *LastByte* member is undefined.
	WC_SIZE	If set, the window size is programmable within the range specified by the *MinSize* and *MaxSize* members. If reset, the window size is fixed to the size indicated by the *MinSize* member. When reset, both the *MinSize* and *MaxSize* members should be the same value.
	WC_WENABLE	If set, the window may be disabled and enabled without reprogramming its characteristics. If reset, the client must preserve window state information before disabling the window.
	WC_8BIT	If set, the window may be programmed for 8-bit data bus width. If reset, the window may not be used for 8-bit data transfers.
	WC_16BIT	If set, the window may be programmed for 16-bit data bus width. If reset, the window may not be used for 16-bit data transfers.
	WC_BALIGN	If set, the window base address must be programmed to align with a multiple of the window size. For example, a window 16 KBytes in size needs to start on a 16 KByte boundary in the host system memory address space. If reset, the window base address may be programmed anywhere in the window's valid range, subject to any constraint specified by *ReqBase*.
	WC_POW2	If set, a window with WC_SIZE also set must be sized between the *MinSize* and *MaxSize* members as a power of two of the *ReqGran* member. If reset, a window with WC_SIZE set may be any multiple of the *ReqGran* member between the *MinSize* and *MaxSize* members. For example, if *ReqGran* is 4 KBytes, *MinSize* is 4 KBytes, *MaxSize* is 64 KBytes and WC_POW2 is set, the possible window sizes are 4, 8, 16, 32 and 64 KBytes. If WC_POW2 is reset, possible windows sizes include all sixteen multiples of 4 KBytes between 4 and 64 KBytes.

	WC_CALIGN	If set, 16-bit PC Card offsets are required to be specified to **SetPage** in increments of the size of the window. If reset, 16-bit PC Card offsets may be specified to **SetPage** without relation to the size of the window. For example, if WC_CALIGN is set and the window is 16 KBytes in size, all 16-bit PC Card offsets specified to **SetPage** must be on 16 KByte boundaries.
	WC_PAVAIL	If set, the window has hardware available which is capable of dividing the window into multiple pages. If reset, the entire window must be addressed as a single page.
	WC_PSHARED	If set, a window's paging hardware is shared with another window. A request to use the paging hardware may fail if the other window is using the paging hardware. If reset, the window's paging hardware is dedicated and a request to use the paging hardware should never fail. This value is only valid if WC_PAVAIL is set. A Socket Services client should check WC_PSHARED if intending to use paging services. If set, the client must ensure that a subsequent **SetWindow** request requiring paging hardware succeeds before attempting to utilize the window as the paging hardware may have already been assigned to another window. To determine if the pager is available, attempt to assign it to a window using **SetWindow** and check for successful return status from the request.
	WC_PENABLE	If set, the page may be disabled and enabled without reprogramming its characteristics. If reset, the client must preserve page state information before disabling the page.
	WC_WP	If set, the window may be write-protected to prevent writing 16-bit PC Card memory mapped into host system memory address space. If reset, the window may not be write-protected to prevent writing 16-bit PC Card memory mapped into host system memory address space. Write-protection is enabled and disabled with the **SetPage** service which requires this support to be available on a page basis for windows which have multiple pages.
FirstByte	First byte addressable in host system memory address space by window. If window *Base* is not programmable, this is the fixed base address of the window.	
LastByte	Last byte addressable in host system memory address space by window. The last byte of the window (base address programmed plus window size minus one) may not exceed this value. If window *Base* is not programmable, this member is undefined. If *LastByte* is expressed in units other than bytes, any address bits of lesser significance not directly expressed are assumed to be set to one (1). For example, if *LastByte* is expressed in 4 KByte units, a value of A3H indicates the last addressable byte within the window is at location A3FFFH in the host system's memory address space.	
MinSize	The minimum window size. When window size is programmed with **SetWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements.	
MaxSize	The maximum window size. When window size is programmed with **SetWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements. The window size may be further limited by the base address of the window. The base address plus the window size minus one must not exceed the *LastByte* member for windows with programmable sizes. If *MaxSize* is zero, window size is the largest value that may be represented by the **SIZE** data type plus one.	

ReqGran	This member describes the required units for expressing window size due to hardware constraints. If the window size is fixed (WC_SIZE is reset), this member will be the same as the *MinSize* and *MaxSize* members.
ReqBase	If WC_BALIGN is reset, this member describes any alignment boundary requirement for programming the window's base address with **SetWindow**.
	If WC_BALIGN is set, this field is undefined.
ReqOffset	If WC_CALIGN is reset, this member describes any alignment boundary requirement for programming the PC Card offset address with **SetPage**.
	If WC_CALIGN is set, this field is undefined.
Slowest	This member represents the slowest access speed supported by the window.
Fastest	This member represents the fastest access speed supported by the window.

Comments

The *Slowest* and *Fastest* members use the format of the Device Speed Code and Extended Device Speed Codes of the Device Information Tuple. (See the ***Metaformat Specification***.) For Socket Services, Bit 7 of the *Slowest* and *Fastest* members is reserved and is reset to zero (0).

The Device Speed Code values are used when what would be the mantissa of an Extended Device Speed Code is reset to zero (0). If the mantissa is non-zero, supported device speeds are coded according to the Extended Device Speed Code. (See the ***Metaformat Specification***.)

Memory windows map accesses to host system memory address space into accesses to memory address space located on a PC Card. How the socket hardware performs this mapping determines the memory characteristics table definition. While memory windows are described by a number of characteristics, most window mapping hardware falls into one of two categories with each category having a single set of characteristics.

Direct window mapping hardware selects a fixed combination of high order address lines (typically via mask and match registers) on the PC Card whenever an access is made within the host system memory address range assigned to the window. Low order address lines are routed directly to the PC Card.

The window size determines how many low order address lines are routed directly to the PC Card. The fixed combination used for the high order address lines is set by the **SetPage** service. This type of window mapping hardware requires the window size be a power of two and that the base address be aligned on a multiple of the window size since mapping is related to the number of low order address lines routed directly to the PC Card.

Translating window mapping hardware uses additional logic to compute a PC Card address. When an access is made to a location within the host system address range mapped by the window, the hardware computes the offset of this location from the beginning of the mapped range (typically via base and length registers) and adds it to the starting offset on the PC Card as set by the **SetPage** service.

While high order address lines may still be set to a fixed combination and some number of low order address lines may be directly routed to the PC Card, mid-order address lines are computed by the window mapping hardware. This type of hardware does not require the window be sized as a power of two or aligned on a boundary related to the window size. However, the window size must be a multiple of the *ReqGran* field.

In summary, if direct window mapping hardware is used, the WC_BALIGN, WC_POW2 and WC_CALIGN parameters are set and the *ReqBase* and *ReqOffset* members are not used. If translating window hardware is used, the WC_BALIGN, WC_POW2 and WC_CALIGN parameters are reset and the *ReqBase* and *ReqOffset* members are significant.

The *ReqBase*, *ReqOffset* and *ReqGran* members are related to the number of low order address lines which are routed directly to the PC Card. For example, if the twelve (12) least significant address lines are routed directly to the PC Card, the *ReqGran* member will indicate the window must be sized as a multiple of 4 KBytes. If translating window hardware is used, the *ReqBase* and *ReqOffset* will also indicate the requirement to align the window base address and the PC Card offset on a 4 KByte boundary.

The following table illustrates the relationship of **Memory Window Characteristics Table** members to the type of hardware used to implement the window:

Member/Parameter	Direct	Translating
WC_BALIGN	Set	reset
WC_POW2	Set	reset
WC_CALIGN	Set	reset
ReqGran	Significant	Significant
ReqBase	Not Used	Significant
ReqOffset	Not Used	Significant

I/O Window Characteristics Table

```
typedef struct tagIOWINTBL {
      FLAGS16  IOWndCaps;
      BASE     FirstByte;
      BASE     LastByte;
      SIZE     MinSize;
      SIZE     MaxSize;
      SIZE     ReqGran;
      COUNT    AddrLines;
      FLAGS8   EISASlot;
} IOWINTBL;
```

Member	Description	
IOWndCaps	Flags indicating I/O window characteristics. This member can be a combination of the following values:	
	Value	**Meaning**
	WC_BASE	If set, the base address of the window is programmable within the range specified by the *FirstByte* and *LastByte* members. If reset, the base address of the window is fixed in host system I/O address space at the address specified in the *FirstByte* member. When WC_BASE is reset, the *LastByte* member is undefined.
	WC_SIZE	If set, the window size is programmable within the range specified by the *MinSize* and *MaxSize* members. If reset, the window size is fixed to the size indicated by the *MinSize* member. When WC_SIZE is reset, both the *MinSize* and *MaxSize* members should be the same value.
	WC_WENABLE	If set, the window may be disabled and enabled without reprogramming its characteristics. If reset, the client must preserve window state information before disabling the window.
	WC_8BIT	If set, the window may be programmed for 8-bit data bus width. If reset, the window may not be used for 8-bit data transfers.
	WC_16BIT	If set, the window may be programmed for 16-bit data bus width. If reset, the window may not be used for 16-bit data transfers.
	WC_BALIGN	If set, the window base address must be programmed to align with a multiple of the window size. For example, an 8 byte window needs to start on an 8 byte boundary in the host system I/O address space. If reset, the window base address may be programmed anywhere in the window's valid range, subject to any constraint specified by *ReqBase*.
	WC_POW2	If set, a window with WC_SIZE set must be sized between the *MinSize* and *MaxSize* members as a power of two of the *ReqGran* member. If reset, a window with WC_SIZE set may be any multiple of the *ReqGran* member between the *MinSize* and *MaxSize* members. For example, if ReqGran is 4 bytes, *MinSize* is 4 bytes, *MaxSize* is 64 bytes and WC_POW2 is set, the possible window sizes are 4, 8, 16, 32 and 64 bytes. If WC_POW2 is reset, possible windows sizes include all sixteen multiples of 4 bytes between 4 and 64 bytes.

	WC_INPACK	If set, the window supports the **INPACK#** signal from a PC Card. This signal allows I/O windows to overlap in the host system's I/O address space.
		If reset, the **INPACK#** signal from a PC Card is ignored by the window hardware. In this case, I/O windows may not overlap in the host system's I/O address space.
	WC_EISA	If set, the window supports I/O mapping in a the same manner as host systems with EISA buses. The *EISASlot* member describes the slot-specific address decodes for this window.
		If reset, the window does not support EISA-like I/O mapping.
	WC_CENABLE	If set, EISA-like common address space enables may be programmed to be ignored.
		If reset, if the window is programmed for EISA-like I/O mapping, the PC Card will receive a card enable signal whenever an access is made to an EISA common address.
		This value is only valid if WC_EISA is set.
FirstByte	First byte addressable in host system I/O address space by window. If window base is not programmable, this is the fixed base address of the window.	
LastByte	Last byte addressable in host system I/O address space by window. The last byte of the window (base address programmed plus window size minus one) may not exceed this value.	
	If window base is not programmable, this member is undefined.	
	If *LastByte* is expressed in units other than bytes, any address bits of lesser significance not directly expressed are assumed to be set to one (1). For example, if *LastByte* is expressed in 4 KByte units, a value of A3H indicates the last addressable byte within the window is at location A3FFFH in the host system's I/O address space.	
MinSize	The minimum window size. When window size is programmed with **SetWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements.	
MaxSize	The maximum window size. When window size is programmed with **SetWindow** it must lie in the range of the *MinSize* and *MaxSize* members and meet all granularity and base requirements.	
	The window size may be further limited by the base address of the window. The base address plus the window size minus one must not exceed the *LastByte* member for windows with programmable sizes.	
	If *MaxSize* is zero, window size is the largest value that may be represented by the **SIZE** data type plus one.	
ReqGran	This member describes the required units for expressing window size due to hardware constraints. If the window size is fixed (WC_SIZE is reset), this member will be the same as the *MinSize* and *MaxSize* members.	
AddrLines	Number of address lines decoded by window. Typically ten (10) or sixteen (16). If a window only decodes ten address lines, accesses to locations above 1 KByte will drive card enables to a PC Card when the ten least significant address lines fall within the range defined by the base address and window size.	
EISASlot	Upper byte used for window-specific EISA I/O address decoding. Describes the upper four address lines used to determine EISA slot-specific addresses used to drive card enables.	
	This member is undefined if WC_EISA is reset.	

5.3.21 PauseEDC [BOTH]

RETCODE = **PauseEDC** (*Adapter*, *EDC*)
ADAPTER *Adapter*;
EDC *EDC*;

The **PauseEDC** service pauses EDC generation on a configured and computing EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

This service is used to pause EDC generation so some accesses to a PC Card are not involved in the computation of an EDC value. This service is only supported if EC_PAUSABLE is set in the **InquireEDC** *Caps* parameter.

See Also **InquireEDC**, **GetEDC**, **SetEDC**, **StartEDC**, **ResumeEDC**, **StopEDC**, **ReadEDC**

5.3.22 ReadEDC [BOTH]

RETCODE = ReadEDC (*Adapter, EDC, Value*)
ADAPTER *Adapter*;
EDC *EDC*;
DWORD *Value*;

The **ReadEDC** service reads the EDC value computed by the EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.
Value	O	Returns computed EDC value. If the generator was set to ET_CHECK8, only the low byte is significant. If the generator was set to ET_SDLC16, only the low word is significant. If the generator was set to ET_SDLC32, all 32-bits are significant.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

If the generator has been used inappropriately (generator not assigned a socket or a combination of reads and writes were used), the computed *Value* may be erroneous.

See Also **InquireEDC**, **GetEDC**, **SetEDC**, **StartEDC**, **PauseEDC**, **ResumeEDC**, **StopEDC**

5.3.23 ResetSocket [BOTH]

RETCODE = **ResetSocket** (*Adapter*, *Socket*)
ADAPTER *Adapter*;
SOCKET *Socket*;

The **ResetSocket** service resets the PC Card in the socket and returns socket hardware to its power-on default state.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.

Return Codes

SUCCESS	if *Adapter* and *Socket* are valid and there is a PC Card in the socket
BAD_ADAPTER	if *Adapter* is invalid
BAD_SOCKET	if *Socket* is invalid
NO_CARD	if there is no PC Card in the socket

Comments

This service toggles the **RESET** pin of the card in the specified socket on the specified adapter.

This service completes an entire RESET pulse, toggling the pin to the RESET state and back to the normal state. It ensures the minimum RESET pulse width is observed. It does NOT wait after returning the **RESET** pin to its normal state. The client must ensure that a card is not accessed before it is **READY** after this service has returned.

All socket hardware is returned to its default power-on state:

- *IFType* set to IF_MEMORY if a 16-bit PC Card is in the socket or to *IF_CARDBUS* if a CardBus PC Card is in the socket.
- *IREQRouting* disabled.
- **VCC**, **VPP1** and **VPP2** set to 5 V DC for a 5 volt only system, otherwise set to the voltage specified by the **VS1#** and **VS2#** pins.
- All windows, pages and EDC generators disabled.

5.3.24 ResumeEDC [BOTH]

RETCODE = ResumeEDC (*Adapter*, *EDC*)
ADAPTER *Adapter*;
EDC *EDC*;

The **ResumeEDC** service resumes EDC generation on a configured and paused EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

This service is used to resume EDC generation so accesses to a PC Card are involved in the computation of an EDC value. This service is only supported if EC_PAUSABLE is set in the **InquireEDC** *Caps* parameter.

See Also **InquireEDC**, **GetEDC**, **SetEDC**, **StartEDC**, **PauseEDC**, **StopEDC**, **ReadEDC**

5.3.25 SetAdapter [BOTH]

RETCODE = **SetAdapter** (*Adapter, State, SCRouting*)
ADAPTER *Adapter*;
FLAGS8 *State*;
IRQ *SCRouting*;

The **SetAdapter** service sets the configuration of the specified adapter.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
State	I	Requested state of the adapter hardware. This parameter can be a combination of the following values:

Value	Meaning
AS_POWERDOWN	If set, adapter hardware should attempt to conserve power. Before an adapter conserving power may be used, full power must be restored using this service. If reset, adapter hardware should enter the fully-powered, fully functional state.
AS_MAINTAIN	If set, all adapter and socket configuration information is maintained while power consumption is reduced. If reset, adapter and socket configuration information must be maintained by the client. This value is only valid if the AS_POWERDOWN value is set.

SCRouting	I	Sets status change interrupt routing. The routing level and active-state are validated even if routing is being disabled. This parameter is an **IRQ** data type. It is a combination of a binary value representing the interrupt level the status change interrupt is currently routed to and the following optional bit-masks:

Value	Meaning
IRQ_HIGH	If set, status change interrupt is set to be active-high. If reset, status change interrupt is set to active-low. On adapters that do not have programmable status change level logic, the desired interrupt level must match the actual hardware or the request is failed returning BAD_IRQ.
IRQ_ENABLE	If set, status change interrupt is enabled. If an unmasked status change event occurs, the adapter generates a hardware interrupt of the specified level. If reset, status change interrupts are not generated by the adapter.

Return Codes

SUCCESS	if *Adapter* is valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_IRQ	if *StatusChange* specifies an unsupported *State* or IRQ level

Comments

Preserving state information may not allow the same level of power reduction as not preserving state information. The ability to reduce power consumption is vendor specific and reduced power settings may not result in any power savings. For example, if an adapter supports a reduced power consumption mode, but is unable to preserve state information in that mode, requests for reduced

power consumption and state preservation may be ignored and SUCCESS returned. The actual adapter configuration is returned by the **GetAdapter** request.

All parameters have been designed to map directly to the values returned by the **GetAdapter** service. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetAdapter**, make changes and then use this service to modify the configuration without having to create initial values for each parameter.

See Also **InquireAdapter**, **GetAdapter**

5.3.26 SetBridgeWindow [BOTH]

RETCODE = SetBridgeWindow (*Adapter, Window, Socket, Size, State, Base)*

ADAPTER *Adapter*;
WINDOW *Window*;
SOCKET *Socket*;
SIZE *Size*;
FLAGS8 *State*;
BASE *Base*;

The **SetBridgeWindow** service sets the current configuration of the bridge window specified by the input parameters. If present on the adapter, PC Card bridge windows are required to allow access to devices on PC Cards.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Specifies a bridge window on the adapter.
Socket	I	Sets physical socket the bridge window is currently assigned.
Size	I	Sets the size of the bridge window in bytes.
State	I	Defined as below. Sets the state of the bridge window hardware. This parameter can be a combination of the following values:
		Value / **Meaning**
		WS_IO — If set, this bridge window routes host system I/O accesses to the PC Card socket.
		If reset, this bridge window routes host system memory accesses to the PC Card socket.
		WS_ENABLED — If set, the bridge window is enabled and routing host system accesses to a PC Card socket.
		If reset, the bridge window is disabled.
Base	O	Sets the base address of the specified bridge window. It is the first address within the host system memory or I/O address space routed to the PC Card socket.

Return Codes:

SUCCESS	if all parameters are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_ATTRIBUTE	if requested *State* does not match the window's capabilities
BAD_BASE	if the *Base* is invalid
BAD_SIZE	if *Size* is invalid
BAD_SOCKET	if *Socket* is invalid for Window
BAD_TYPE	if WS_IO setting is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values returned by the **GetBridgeWindow** service. This is intended to allow clients of Socket Services to retrieve current configuration information with this service, make changes and then use the **SetBridgeWindow** service to modify the configuration without having to create initial values for each parameter.

See Also **InquireBridgeWindow**, **GetBridgeWindow**, **InquireWindow**, **GetWindow**, **SetWindow**, **AccessConfigSpace**.

5.3.27 SetEDC [BOTH]

RETCODE = **SetEDC** (*Adapter, EDC, Socket, State, Type*)

ADAPTER	*Adapter*;
EDC	*EDC*;
SOCKET	*Socket*;
FLAGS8	*State*;
FLAGS8	*Type*;

The **SetEDC** service sets the configuration of the EDC generator specified by the input parameters.

Parameter	I/O	Description	
Adapter	I	Specifies a physical adapter on the host system.	
EDC	I	Specifies a physical EDC generator on the adapter.	
Socket	I	Specifies the physical socket on the adapter that the EDC generator is to be assigned.	
State	I	Sets the current state of the EDC generator. This field may be combination of the following values:	
		Value	**Meaning**
		EC_UNI	If set, EDC generator is computes in only one direction. EC_WRITE determines whether computation is on read or write accesses.
			If reset, EDC generator is computes on both read and write accesses.
		EC_WRITE	If set, EDC generator is computes only on write accesses.
			If reset, EDC generator is computes only on read accesses.
			This value is only valid if EC_UNI is set.
Type	I	Sets type of EDC generated. This parameter may be one of the following values:	
		Value	**Meaning**
		ET_CHECK8	EDC generated is 8-bit checksum.
		ET_SDLC16	EDC generated is 16-bit CRC-SDLC.
		ET_SDLC32	EDC generated is 32-bit CRC-SDLC.

Return Codes

SUCCESS	if *Adapter*, *EDC*, *Socket*, *State* and *Type* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_ATTRIBUTE	if *State* or *Type* is invalid
BAD_EDC	if *EDC* is invalid
BAD_SOCKET	if *Socket* is invalid

Comments

All parameters have been designed to map directly to the values returned by the **GetEDC** service. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetEDC**, make changes and then use this service to modify the configuration without having to create initial values for each parameter.

See Also **InquireEDC**, **GetEDC**, **StartEDC**, **PauseEDC**, **ResumeEDC**, **StopEDC**, **ReadEDC**

5.3.28 SetPage [PC16]

RETCODE = SetPage (*Adapter, Window, Page, State, Offset*)

ADAPTER *Adapter*;
WINDOW *Window*;
PAGE *Page*;
FLAGS8 *State*;
OFFSET *Offset*;

The **SetPage** service configures the page specified by the input parameters. It is only valid for memory windows (WS_IO is reset for the *Window*). This service is unsupported by CardBus PC Card.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Specifies a physical window on the adapter.
Page	I	Specifies the page within the *Window*.
State	I	Programs the state of the *Page* within the *Window*. This parameter can be a combination of the following values:

Value	Meaning
PS_ATTRIBUTE	If set and *Page* is enabled, page is programmed to map PC Card attribute memory into host system memory space. If reset and *Page* is enabled, page is programmed to map PC Card common memory into host system memory space.
PS_ENABLED	If set, *Page* is enabled and maps PC Card memory into the host system memory or I/O space. If reset, *Page* is disabled. Some hardware implementation may not allow individual pages to be disabled, only entire windows. If there is only a single page in the window, the window is disabled by this request. This request returns BAD_ATTRIBUTE for multi-paged windows if the pages cannot be individually disabled.
PS_WP	If set, *Page* is write-protected by page mapping hardware in socket. If reset, *Page* is not write-protected by socket's page-mapping hardware. However, the PC Card memory may be write-protected in other ways. If set and the window does not support write-protection, BAD_ATTRIBUTE is returned.

Offset	I	The offset of a PC Card's memory to be mapped into host system memory space by this page. The following formula may be used to calculate the system memory address to access the PC Card memory being mapped by the page: Base + (Page * 16 KBytes)

Return Codes

SUCCESS	if *Adapter*, *Offset*, *Page*, *State* and *Window* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_ATTRIBUTE	if *State* is invalid
BAD_OFFSET	if *Offset* is invalid
BAD_PAGE	if *Page* is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values returned by the **GetPage** service. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetPage**, make changes and then use this service to modify the configuration without having to create initial values for each parameter.

All pages in windows which are subdivided into multiple pages are 16 KBytes in size. A window with only a single page may be any size meeting the constraints returned by **InquireWindow**.

To map PC Card memory into system memory requires that both the WS_ENABLED value of the *State* field used by **Get/SetWindow** be set and the PC_ENABLED value of the *State* field used by **Get/SetPage** be set. For windows with WS_PAGED reset, the PS_ENABLED value is ignored by **SetPage**. The window is enabled and disabled by the WS_ENABLED value of **SetWindow**. **GetPage** for windows with WS_PAGED reset reports the value of WS_ENABLED for PS_ENABLED.

For windows with WS_PAGED set, WS_ENABLED acts as a global enable / disable for all pages within the window. Once WS_ENABLED has been set using **SetWindow**, individual pages may be enabled and disabled using **SetPage** and PS_ENABLED.

If WC_WENABLE is reported as set by **InquireWindow**, Socket Services preserves the state of PS_ENABLED for each page in the window whenever WS_ENABLED is changed by **SetWindow**. If WC_ENABLE is reported as reset by **InquireWindow**, the client must use **SetPage** to set the PS_ENABLED state for each page within the window after WS_ENABLED is set with **SetWindow**.

See Also **InquireWindow**, **GetWindow**, **SetWindow**, **GetPage**

5.3.29 SetSocket [BOTH]

RETCODE = **SetSocket** (*Adapter, Socket, SCIntMask, Vcontrol, VccLevel, VppLevels, State, CtlInd, IREQRouting, IFType, IFIndex*)

ADAPTER	*Adapter*;
SOCKET	*Socket*;
FLAGS8	*SCIntMask*;
PWRINDEX	*Vcontrol*;
PWRINDEX	*VccLevel*;
PWRINDEX	*VppLevels*;
FLAGS8	*State*;
FLAGS8	*CtlInd*;
IRQ	*IREQRouting*;
FLAGS8	*IFType*;
WORD	*IFIndex;*

The **SetSocket** service sets the current configuration of the socket identified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Socket	I	Specifies a physical socket on the adapter.
SCIntMask	I	Sets mask for events that generate a status change interrupt when they occur on the socket. If a value is set the event generates a status change interrupt if the following conditions are met: The event is supported as indicated by the *SCIntCaps* parameter of **InquireSocket** and status change interrupts have been enabled by **SetAdapter**. This parameter is a combination of the SBM_x values defined in **InquireSocket**. An attempt to set an event that is unsupported by *SCIntCaps* will not return an error. **GetSocket** will only return values that are supported by *SCIntCaps*.
Vcontrol	I	This parameter takes on the following values:

Value	Meaning
VCTL_CISREAD	If reset, the **VCC** level and **VPP[2::1]** levels are controlled by the *VccLevel* and *VppLevels* fields. If set, the **VCC** level and **VPP[2::1]** levels are set to the value indicated by the voltage sense signaling from the PC Card and the *VccLevel* and *VppLevels* fields are ignored.
VCTL_OVERRIDE	VCTL_OVERRIDE applies only to 16-bit PC Cards. The CardBus PC Card interface requires the **VCC** level match the value indicated by the voltage sense signaling from the PC Card or this service returns an error. If reset, the **VCC** level must match the value indicated by the voltage sense signaling from the PC Card or this service returns an error. If set, the *VccLevel* need not match the value indicated by the voltage sense signaling from the PC Card. If the *VccLevel* does match the voltage sense signaling from the PC Card, VCTL_OVERRIDE is reset when returned by **GetSocket**.

Note: The VCTL_CISREAD and VCTL_OVERRIDE bits are mutually exclusive. If both are set, BAD_ATTRIBUTE is returned. In addition, Socket Services will reset these bits upon card removal.

Note: When initially powering a PC Card with SetSocket, if the VCTL_CISREAD value is set, the PC Card is powered to the value indicated by the voltage sense signaling from the card.

VccLevel I Sets current power level of **Vcc** signal. This is an index into the array of **PWRENTRY** items returned by **InquireAdapter**. Valid values range from zero to one less than the number of levels returned by **InquireAdapter**.

On Low Voltage capable systems Socket Services must observe that state of the **VS1#** and **VS2#** pins when requesting voltage changes on the **Vcc** and **Vpp[2::1]** pins to prevent inappropriate voltages being applied to the card unless VCTL_OVERRIDE is set. The VCTL_OVERRIDE is provided to protect the card from a Low Voltage unaware client. Proper procedure must be used to determine appropriate voltage levels for the card, this includes assuring that systems that are not X.X V capable must be X.X V aware to ensure that X.X V cards are not damaged. (See the ***Electrical Specification***.)

When changing from one non-zero **Vcc** to another non-zero **Vcc**, Socket Services is required to observe a Power-up/Power-down timing sequence. (See the ***Electrical Specification***.)

VppLevels I Sets current power level of **Vpp[2::1]** signals. This is two indices into the array of **PWRENTRY** items returned by **InquireAdapter**. Separate values are in this parameter for the **Vpp1** and **Vpp2** signals. Valid values for each index range from zero to one less than the number of levels returned by **InquireAdapter**.

Note: The *VccLevel* and *VppLevels* always return the actual levels currently applied to the card.

State I Resets latched values representing state changes experienced by the socket hardware. Only those values set in the **InquireSocket** *SCRptCaps* parameter are supported. Attempts to reset unsupported values are ignored.

This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *SCIntCaps* and *SCRptCaps* parameters.

CtlInd I Sets socket controls and indicators. If a value is set, the corresponding control or indicator is turned-on. If a value is reset, the corresponding control or indicator is turned-off. Values supported by the socket are defined by the *CtlIndCaps* parameter returned by **InquireSocket**.

This parameter is a combination of the SBM_x values defined in **InquireSocket** for the *CtlIndCaps* parameter.

IREQRouting I Sets PC Card **IREQ#** routing. This parameter is an **IRQ** data type.

This parameter is ignored if *IFType* is not IF_IO or IF_CARDBUS. If *IFType* is IF_IO or IF_CARDBUS, routing level and inverter state are validated even if routing is being disabled.

This parameter is a combination of a binary value representing the IRQ level used for routing the PC Card **IREQ#** signal and the following optional values:

Value	Meaning
IRQ_INVALID	If set, the binary value representing the IRQ level is invalid and should be ignored. This bit may not be set with IRQ_HIGH.
	If reset, the binary value representing the IRQ level is valid.
IRQ_HIGH	If set, the PC Card **IREQ#** signal is inverted.
	If reset, the PC Card **IREQ#** signal is routed without inversion.
IRQ_ENABLE	If set, **IREQ#** routing is enabled.
	If reset, **IREQ#** routing is not enabled and interrupts from a PC Card in the socket are ignored.

IFType I Sets the current interface type. Uses the same definitions as the *IFType* parameter of **GetSocket**.

When a CardBus PC Card is inserted in a socket, this field is ignored. Sockets automatically configure to IF_CARDBUS when a CardBus PC Card is inserted.

IFIndex I Sets the Custom Interface setting when *IFType* is set to IF_CUSTOM. This is an index into the array of *dCustomIF* items returned by **InquireAdapter**. Valid values range from zero to one less than the number of interface numbers returned by **InquireAdapter**.

This field is ignored when *IFType* is not set to IF_CUSTOM.

Return Codes

SUCCESS	if *Adapter* and *Socket* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_IRQ	if *IREQRouting* not supported
BAD_SOCKET	if *Socket* is invalid
BAD_TYPE	if *IFType* not supported
BAD_VCC	if **VCC** level is invalid
BAD_VPP	if **VPP1** or **VPP2** level is invalid
BAD_ATTRIBUTE	if both CCTL_CISREAD and VCTL_OVERRIDE are set

Comments

All parameters have been designed to map directly to the values returned by the **GetSocket** service. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetSocket**, make changes and then use this service to modify the configuration without having to create initial values for each parameter.

See Also **InquireSocket**, **GetSocket**

5.3.30 SetWindow [PC16]

RETCODE = **SetWindow** (*Adapter, Window, Socket, Size, State, Speed, Base*)

ADAPTER *Adapter*;
WINDOW *Window*;
SOCKET *Socket*;
SIZE *Size*;
FLAGS8 *State*;
SPEED *Speed*;
BASE *Base*;

The **SetWindow** service sets the configuration of the window specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
Window	I	Window number. Specifies a physical window on the adapter. May refer to either a hardware or an adapter window.
Socket	I	Assigns the *Window* to the specified socket. Socket numbers range from zero to fifteen using bits 0 to 3. The rest of the bits in this field are binding specific.
Size	I	Sets the window's size. If *Size* is equal to zero (0), the window is the maximum size that may be represented by the data type used for this parameter plus one. For example, if the data type used for *Size* is a word and it is expressed in units of a byte, a value of zero represents a window size of 65,536 bytes.
State	I	Sets the state of the window hardware as defined as below. This parameter can be a combination of the following values:

Value	Meaning
WS_IO	If set, window maps registers on a 16-bit PC Card into the host system's I/O address space.
	If reset, window maps memory address space on a 16-bit PC Card into the host system's memory address space.
WS_ENABLED	If set, window is enabled and mapping a 16-bit PC Card into the host system memory or I/O address space.
	If reset, window is disabled.
WS_16BIT	This value is only valid for 16-bit PC Cards.
	If set, window is programmed for a 16-bit data bus width.
	If reset, window is programmed for an 8-bit data bus width.
WS_PAGED	If set, window is subdivided into multiple 16 KByte pages whose card offset addresses may be set individually using **SetPage**.
	If reset, window is a single page.
	This value is only valid for memory windows (WS_IO reset) on 16-bit PC Cards.
WS_EISA	If set, window is using EISA I/O mapping.
	If rest, window is using ISA I/O mapping.
	This value is only valid for I/O windows (WS_IO set).
WS_CENABLE	If set, accesses to I/O ports in EISA common I/O areas generate card enables.
	If reset, accesses to I/O ports in EISA common I/O areas are ignored.
	This value is only valid for I/O windows (WS_IO set) that have WS_EISA set.

Speed	I	This parameter is the access speed the client wishes to use for the window. It uses the format of the Device Speed Code and Extended Device Speed Codes of the Device Information Tuple. (See the ***Metaformat Specification*** and the ***Electrical Specification***.) If Socket Services does not support the speed requested, it uses the next slowest speed it supports. For Socket Services, Bit 7 of the *Speed* is reserved and is reset to zero (0). This parameter is ignored for I/O windows (WS_IO set).
Base	I	Programs the base address of the specified window. It is the first address within the system memory or I/O space to which the window responds.

Return Codes

SUCCESS	if all parameters are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_ATTRIBUTE	if requested *State* does not match the window's capabilities
BAD_BASE	if the *Base* is invalid
BAD_SIZE	if *Size* is invalid
BAD_SOCKET	if *Socket* is invalid for *Window*
BAD_SPEED	if *Speed* is too slow
BAD_TYPE	if WS_IO setting is invalid
BAD_WINDOW	if *Window* is invalid

Comments

All parameters have been designed to map directly to the values returned by the **GetWindow** service. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetWindow**, make desired changes and then use this service to modify the configuration without having to create initial values for each parameter.

The following comments apply to 16-bit PC Card only:

- For memory mapping windows, the area of the PC Card memory array mapped into the host system memory space is managed by **GetPage** and **SetPage** requests.
- To map PC Card memory address space into host system memory address space requires that both the WS_ENABLED value of the *State* parameter used by **Get/SetWindow** be set and the PC_ENABLED value of the *State* parameter used by **Get/SetPage** be set. For windows with WS_PAGED reset, the PS_ENABLED value is ignored by **SetPage**. The window is enabled and disabled by the WS_ENABLED value of **SetWindow**. **GetPage** for windows with WS_PAGED reset reports the value of WS_ENABLED for PS_ENABLED.
- For windows with WS_PAGED set, WS_ENABLED acts as a global enable / disable for all pages within the window. Once WS_ENABLED has been set using **SetWindow**, individual pages may be enabled and disabled using **SetPage** and PS_ENABLED.
- If WC_WENABLE is reported as set by **InquireWindow**, Socket Services preserves the state of PS_ENABLED for each page in the window whenever WS_ENABLED is changed by **SetWindow**. If WC_ENABLE is reported as reset by **InquireWindow**, the client must use **SetPage** to set the PS_ENABLED state for each page within the window after WS_ENABLED is set with **SetWindow**.

See Also **InquireWindow**, **GetWindow**, **GetPage**, **SetPage**, **InquireBridgeWindow**, **GetBridgeWindow**, **SetBridgeWindow**

5.3.31 StartEDC [BOTH]

RETCODE = **StartEDC** (*Adapter, EDC*)
ADAPTER *Adapter*;
EDC *EDC*;

The **StartEDC** service starts a previously configured EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

Comments

This service loads the EDC generator with any required initialization value to properly compute the configured type of EDC.

See Also **InquireEDC**, **GetEDC**, **SetEDC**, **PauseEDC**, **ResumeEDC**, **StopEDC**, **ReadEDC**

5.3.32 StopEDC[BOTH]

RETCODE = StopEDC (*Adapter, EDC*)
ADAPTER *Adapter*;
EDC *EDC*;

The **StopEDC** service stops EDC generation on a configured and computing EDC generator specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.
EDC	I	Specifies a physical EDC generator on the adapter.

Return Codes

SUCCESS	if *Adapter* and *EDC* are valid
BAD_ADAPTER	if *Adapter* is invalid
BAD_EDC	if *EDC* is invalid

See Also **InquireEDC**, **GetEDC**, **SetEDC**, **StartEDC**, **PauseEDC**, **ResumeEDC**, **ReadEDC**

5.3.33 VendorSpecific [BOTH]

RETCODE = VendorSpecific (*Adapter* …)
ADAPTER *Adapter*;

This service is vendor specific. The service is reserved for vendors to add proprietary extensions to the Socket Services interface. No guarantee is made that any mode-specific pointer conversion will be handled correctly. Vendors should attempt to use the registers in a non-mode specific manner.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host system.

Return Codes

SUCCESS if parameters are valid

Other return codes are specific to the Socket Services handler.

Comments

This service may have additional parameters that are specific to a particular vendor's handler.

Before using this service, a client should use the **GetVendorInfo** service to confirm the implementer to validate whether the vendor specific services are available.

See Also **GetVendorInfo**

6. USING SOCKET SERVICES

This section describes how various services within Socket Services are intended to be used. This section has been included as an aid to understanding how Socket Services is intended to work. The approaches outlined in this section are for clarification only and may not be required.

6.1 Determining Socket Services Resources

The first task any client of Socket Services performs is determining that Socket Services is installed. The **GetAdapterCount** service is used for presence detection returning *Signature* and *TotalAdapters*. Next, the client verifies that a compatible Socket Services is installed. The client checks compatibility by verifying the Socket Services *Compliance* level returned by **GetSSInfo**. If this *Compliance* level is acceptable, the client may also wish to verify whether the vendor's version is acceptable by checking the ASCIIZ string describing the implementer (*Type* = 0) and the *Release* number returned by **GetVendorInfo**. This last step is optional.

Once an acceptable Socket Services handler has been verified, the client may begin to determine the capabilities of the hardware. **GetAdapterCount** returns the number of adapters present in *NumAdapters*. The client may then call **InquireAdapter** for each adapter to determine its capabilities.

InquireAdapter reports the number of sockets on each adapter, the number of memory or I/O mapping windows available on the adapter, the number of EDC generators and whether certain capabilities are available on individual sockets or are implemented on an adapter basis. For instance, an adapter may support hardware indicators only at the adapter level. If a client sets the indicator for Busy Status on one socket and resets it on another socket, the indicator will be left on since the indicator must represent the state of all sockets.

Different hardware implementations may implement window management differently. **InquireAdapter** reports the total number of windows available on the adapter. However, some windows may only be available on specific sockets. In other implementations they may be assignable to any socket. The **InquireWindow** service is used to determine a specific window's characteristics. The **InquireSocket** service is used to determine each socket's characteristics.

Error Detection capability is determined in the same manner, using the **InquireEDC** service. Error detection generators may be dedicated to a particular socket, or useable with more than one socket.

Higher-level software determining Socket Services capabilities may construct RAM-based tables describing the configuration found. These tables might contain information relating to the assignment of Socket Services resources.

6.2 Status Change Handling

A Socket Services client may note status changes in two ways. First, the client may poll Socket Services on a socket-by-socket basis to determine if a change has occurred. This polling may take place at any time that Socket Services is allows entry. The software may poll at prescribed times (such as before using a Socket Services resource) or as prompted by an external source (such as a timer interrupt).

The second approach is to program one or more sockets to generate an interrupt when a status change occurs. Different hardware implementations may limit the number of interrupts available. In these situations more than one socket may be assigned to the same interrupt. The status change

interrupt handler uses the **AcknowledgeInterrupt** service to determine which socket experienced the status change.

The **AcknowledgeInterrupt** service returns a bit-map representing all of the sockets administered by a particular adapter. If a bit is set, the corresponding socket has a condition that could have caused the interrupt. This bit represents current socket status AND-ed with the socket's status change enables.

The final act of the client's status change interrupt handler is to complete interrupt processing by resetting host hardware to prepare for future interrupts. The handler then returns to the interrupted process concluding interrupt handling. Processing continues in the interrupted routine.

During background processing, outside of the hardware interrupt handler, the Socket Services client polls each socket indicating a change with the **GetStatus** service. Returned values indicate the cause of the interrupt.

6.3 Bus-Expanders or Docking Stations

In some instances, clients may choose to expand the number of PC Card sockets on a host by plugging an expander of some type into a socket. An extension to Socket Services is required to address these additional sockets, if they are to be handled transparently to Socket Services clients.

One approach might be to address these sockets as if they existed on another adapter. Software resident in the host could intercept calls to Socket Services and filter the **GetAdapterCount** service and all services addressing this new 'adapter' and the sockets it contains.

The above approach is only one example. The actual implementation of expanding the number of PC Card sockets using an existing socket is vendor specific.

Docking stations are another situation that is quite similar to bus-expanders. An algorithm for handling hot-dock events is defined in section ***3.6 Docking.***

6.4 Using XIP

eXecute-In-Place (XIP) applications require sockets which support memory-mapped windows. In addition, unlike many other clients of PC Card sockets, XIP applications require exclusive, full-time access to the these resources. Higher-level software that utilizes Socket Services resources must ensure that resources used by XIP are dedicated and are not shared with other applications.

6.5 Power Management

Power Management can be an extremely complex issue within host environments. Socket Services merely provides a means to manipulate the power levels available on an adapter, if they are adjustable in the hardware implementation. Socket Services does not deal with Power Management capabilities available on installed cards. These capabilities are expected to be utilized by card-aware drivers through a higher-level software service.

APPENDIX-A

7. SERVICE CODES

Table 7–1 Service Codes–Numerical Order

Service Code	Value
GetAdapterCount	80H
Reserved for historical purposes	81H–82H
GetSSInfo	83H
InquireAdapter	84H
GetAdapter	85H
SetAdapter	86H
InquireWindow	87H
GetWindow	88H
SetWindow	89H
GetPage	8AH
SetPage	8BH
InquireSocket	8CH
GetSocket	8DH
SetSocket	8EH
GetStatus	8FH
ResetSocket	90H
Reserved for historical purposes	91H–94H
InquireEDC	95H
GetEDC	96H

Service Code	Value
SetEDC	97H
StartEDC	98H
PauseEDC	99H
ResumeEDC	9AH
StopEDC	9BH
ReadEDC	9CH
GetVendorInfo	9DH
AcknowledgeInterrupt	9EH
GetSetPriorHandler	9FH
GetSetSSAddr	A0H
GetAccessOffsets	A1H
AccessConfigurationSpace	A2H
InquireBridgeWindow	A3H
GetBridgeWindow	A4H
SetBridgeWindow	A5H
Reserved for future use	A6H–ADH
VendorSpecific	AEH
Reserved for Card Services	AFH

Note: Reserved entries should not be used. They are reserved for historical purposes, Card Services use, or for future expansion.

Table 7–2 Service Codes — Alphabetic Order

Service Code	Value
AccessConfigurationSpace	A2H
AcknowledgeInterrupt	9EH
GetAccessOffsets	A1H
GetAdapter	85H
GetAdapterCount	80H
GetBridgeWindow	A4H
GetEDC	96H
GetPage	8AH
GetSetPriorHandler	9FH
GetSetSSAddr	A0H
GetSocket	8DH
GetSSInfo	83H
GetStatus	8FH
GetVendorInfo	9DH
GetWindow	88H
InquireAdapter	84H
InquireBridgeWindow	A3H
InquireEDC	95H
InquireSocket	8CH

Service Code	Value
InquireWindow	87H
PauseEDC	99H
ReadEDC	9CH
Reserved for Card Services	AFH
Reserved for future use	A6H–ADH
Reserved for historical purposes	81H–82H
Reserved for historical purposes	91H–94H
ResetSocket	90H
ResumeEDC	9AH
SetAdapter	86H
SetBridgeWindow	A5H
SetEDC	97H
SetPage	8BH
SetSocket	8EH
SetWindow	89H
StartEDC	98H
StopEDC	9BH
VendorSpecific	AEH

Note: Reserved entries should not be used. They are reserved for historical purposes, Card Services use, or for future expansion.

APPENDIX-B

8. RETURN CODES

Table 8–1 Return Codes — Numerical Order

Return Code	Value	Description
SUCCESS	00H	The request succeeded
BAD_ADAPTER	01H	Specified adapter is invalid
BAD_ATTRIBUTE	02H	Specified attribute 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 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
Reserved	0CH	Reserved for historical purposes
BAD_TYPE	0DH	Specified window or interface type is invalid
BAD_VCC	0EH	Specified **VCC** power index is invalid
BAD_VPP	0FH	Specified **VPP1** or **VPP2** power 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
BAD_SERVICE	15H	Service not supported
BAD_MODE	16H	Requested processor mode is not supported
BAD_SPEED	17H	Specified speed is invalid/unavailable
BUSY	18H	Unable to process request at this time - retry later
Reserved	19H - FFH	Reserved for Card Services and future expansion

Note: Return Codes common to Card Services use the same values. Reserved values should not be used. They are reserved for historical purposes, Card Services use, or for future expansion.

Table 8–2 Return Codes — Alphabetic Order

Return Code	Value	Description
BAD_ADAPTER	01H	Specified adapter is invalid
BAD_ATTRIBUTE	02H	Specified attribute is invalid
BAD_BASE	03H	Specified base system memory address is invalid
BAD_EDC	04H	Specified EDC generator is invalid
BAD_SERVICE	15H	Service not supported
BAD_IRQ	06H	Specified IRQ level is invalid
BAD_MODE	16H	Requested processor mode is not supported
BAD_OFFSET	07H	Specified PC Card offset is invalid
BAD_PAGE	08H	Specified page is invalid
BAD_SIZE	0AH	Specified size is invalid
BAD_SOCKET	0BH	Specified socket is invalid
BAD_SPEED	17H	Specified speed is invalid/unavailable
BAD_TYPE	0DH	Specified window or interface type is invalid
BAD_VCC	0EH	Specified **VCC** power index is invalid
BAD_VPP	0FH	Specified **VPP1** or **VPP2** power index is invalid
BAD_WINDOW	11H	Specified window is invalid
BUSY	18H	Unable to process request at this time - retry later
NO_CARD	14H	No PC Card in socket
READ_FAILURE	09H	Unable to complete read request
Reserved	05H	Reserved for historical purposes
Reserved	0CH	Reserved for historical purposes
Reserved	10H	Reserved for historical purposes
Reserved	13H	Reserved for historical purposes
Reserved	19H - FFH	Reserved for Card Services and future expansion
SUCCESS	00H	The request succeeded
WRITE_FAILURE	12H	Unable to complete write request

Note: Return Codes common to Card Services use the same values. Reserved values should not be used. They are reserved for historical purposes, Card Services use, or for future expansion.

APPENDIX-C

9. SOCKET SERVICES BINDINGS

9.1 Overview

A Socket Services binding answers the following three questions for a specific host environment:

How is the presence of Socket Services determined?

How are Socket Services requests made?

How are arguments passed to and from Socket Services?

A specific host environment for a Socket 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 Socket Services binding.

9.2 Presence Detection and Installation Notification

A client determines whether Sockets Services is available in the host environment through a binding specific presence detection mechanism. All bindings specify a method of determining the presence of Socket Services using operations that have well-defined responses whether Socket Services is actually installed or not. A Socket Services client may use the Socket Services request mechanism for presence detection if the binding guarantees a negative response is returned if Socket Services is not installed.

Socket Services handlers may be installed before or after Card Services. If a Socket Services handler is installed before Card Services, Card Services uses the binding specific presence detection mechanism to locate the handler. If a Socket Services handler is installed after Card Services, the Socket Services handler notifies Card Services of its installation using a binding specific method.

9.3 Making Socket Services Requests

Socket 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 Socket Services may be appropriate depending on the host system's environment. Environments which emulate other environments may actually provide more than one method of making a Socket Services request. If a Socket 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 Socket Services binding defines how arguments are passed to and from Socket Services. Depending on the host environment, arguments may be passed in registers, in stack-based packets or even in global data areas. There are a number of possible input arguments to a Socket Services request. These include:

Service	The service that Socket Services is being requested to perform.
Adapter	The hardware which connects a host system bus to 68-pin PC Card sockets.
Window	An area in a host system's memory or I/O address space through which a PC Card may be accessed.
Page	A subdivision of a window. If there is more than one page in a window, all pages are 16 KBytes in size.
Socket	The 68-pin socket a PC Card is inserted in.
Counts	Number of items.
Attributes	Typically a bit-mapped field that describes characteristics.
Data Area	Pointer to a Socket Services data area. Provided to Socket Services by its clients in environments where data pointers must be manufactured by the operating system.
Buffer	Pointer to a data buffer.
Offset	An offset into a PC Card's address space.

Many Socket Services interfaces do not use all of the arguments in every request. If an input argument is not used for a service, but the binding provides for a consistent calling structure, the argument is ignored.

Socket Services interfaces may modify arguments to return information. If Socket Services does not use an argument to return information, it is returned unmodified.

All Socket Services interfaces return *Status*. This is a RETCODE as defined in a previous section. A binding may use the same or an overlapping representation for the *Service* input argument and *Status*.

9.5 Power Management and Indicators

Power management and indicators may be available on a per adapter or per socket basis. To provide a consistent interface, Socket Services provides access to these services on a socket basis. It is expected that a hardware implementation that only provides power management and/or indicator control at the adapter level shall provide a Socket Services handler that manages those resources for the entire adapter based on requests to individual sockets.

Socket Services does indicate whether power management and indicator control is performed at the adapter or socket level. However, by providing only one control point (the socket), a client of Socket Services is not required to provide two types of controlling routines.

9.6 x86 Architecture Binding

9.6.1 Overview

This section describes the Socket Services bindings for x86-based computers using various system bus architectures.

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 types of clients that use Socket Services.

An environment must provide a binding for each type of client it supports. The four types of clients defined by this section are:

DOS real mode clients

OS/2 16-bit protect mode clients

Windows 16-bit protect mode clients

Windows 32-bit protect mode VxD clients

9.6.2 Presence Detection

Before Card Services has been loaded, all Socket Services clients determine the presence of Socket Services by making a Socket Services **GetAdapterCount** request in real-mode. If the request returns with the [CF] set or the *Signature* field is not equal to the ASCII characters 'SS', Socket Services is not installed.

If the **GetAdapterCount** request returns with the [CF] reset and the *Signature* field is set to the ASCII characters 'SS', Socket Services is installed. The Socket Services client then performs real-mode **GetSSInfo** requests to determine how many Socket Services handlers are installed in the host system and which adapters each Socket Services handler is managing. See the **GetSSInfo** service description for details.

For each Socket Services handler located using the **GetSSInfo** request, the Socket Services client performs a real-mode **GetSetSSAddr** request to determine the entry point to use for subsequent Socket Services requests to the handler. Separate **GetSetSSAddr** requests are required for each processor mode used by the Socket Services client. See the **GetSetSSAddr** service description for details.

9.6.3 Installation Notification

If a Socket Services handler is installed after Card Services has loaded, the handler notifies Card Services of its presence using Card Services **AddSocketServices** or **ReplaceSocketServices** requests.

How the Socket Services handler locates Card Services and makes Card Services requests is binding specific. See the ***Card Services Specification*** for details.

9.6.4 Making Socket Services Requests

Until Card Services completes its installation, all Socket Services requests are made by placing the appropriate values in the registers indicated below and performing an INT 1AH in real-mode. If a Socket Services handler was installed prior to Card Services, requests made to the handler after Card Services completes its initialization use the mode-specific entry point returned by **GetSetSSAddr** as described above.

If a Socket Services handler is installed after Card Services, as described in the Installation Notification Section above, Card Services uses the entry point provided with the arguments to the Card Services **AddSocketServices** or **ReplaceSocketServices** requests.

9.6.5 Argument Passing

Two methods are used for passing arguments: directly in the CPU registers and in a packet that is referenced by a binding specific pointer. The CPU register method is the standard method used and is always available for backwards compatibility. The packet method is used with the entry-point retrieved via **GetSetSSAddr** *Subfunc*=04h.

9.6.5.1 CPU Register Interface Usage

The Socket Services interface in the x86 environment passes arguments in registers using the following guidelines:

Entry:

[AH]	Service Desired	
[AL]	*Adapter*	
[BH]	*Window*	
[BL]	*Page* or *Socket*	
[CX]	*Count*	
[DX]	*Attributes*	
[DS]:[(E)SI]	*Data Pointer*	Pointer to Socket Services data area, not required and ignored by real-mode Socket Services requests. Card Services (the Socket Services client) determines the appropriate value for protect-mode requests in one of two ways. For each Socket Services handler installed before Card Services, this value is determined by the information returned by a **GetSetSSAddr** request to the handler made during presence detection operations. For each Socket Services handler installed after Card Services, this value is provided to Card Services by the Card Services **AddSocketServices** or **ReplaceSocketServices** request used by each Socket Services handler to notify Card Services of the handler's presence. For OS/2 and Windows 16-bit protect modes the [DS]:[SI] register pair are a selector:offset pointer to the Socket Services data area. Windows 32-bit protect mode (flat model) VxD clients pass the 32-bit offset of the Socket Services data area in the [ESI] register.
[ES]:[(E)DI]	*Buffer*	Pointer to a data buffer for returning information to client. For real-mode the [ES]:[DI] register pair are a segment:offset pointer to the buffer. For OS/2 and Windows 16-bit protect modes the [ES]:[DI] register pair are a selector:offset pointer to the buffer. For Windows 32-bit protect mode (flat model) VxD clients the [EDI] register is the 32-bit offset of the buffer.
[DI]	*Offset*	Used by **SetPage** to set offset of PC Card's memory mapped into host system memory space. Expressed in 4 KByte units.
	Base	Used by **SetWindow** to specify window's base address in host system address space. I/O window bases are expressed in bytes. Memory windows are expressed in 4 KByte units.

Exit:

[CF]	*Status*		Set = error
			Reset = success
	If [CF] set		
		[AH]	Non SUCCESS Return Code
	else		
		[AH]	SUCCESS Return Code

Please note that these are guidelines used to develop the service interfaces and exceptions have been made for specific services. See the individual service bindings for the x86 architecture.

Whenever possible, the interface preserves the contents of all registers unless they are used to return information. For bit-mapped fields, bits within a field (or register) are numbered beginning with zero. Bit 0 is the least significant bit within the register.

For all services, the [CF] indicates whether the service was successful. If the [CF] is reset on exit, the service was successful. If the [CF] is set on exit, the service failed. The [AH] register always contains a RETCODE on exit. The only exception to this convention is determining the presence of Socket Services with the **GetAdapterCount** service. There is no guarantee of the state of the [CF] or the [AH] register if no Socket Services handler is present.

9.6.5.2 Packet Interface Usage

9.6.5.2.1 Overview

The packet interface passes parameters via a packet that is pointed to by a pointer on the stack. In c-style notation, the entry point for the packet interface is defined as:

```
void      SSPacketEntryPoint     (FAR  *fpArgPacket)
```

Note: The usage of a C-style function definition provides processor and mode independence in the definition. Implementers using the packet interface must take care in utilizing the correct mode.

For all packet entry point modes the parameters are passed in a packet that is generally structured as shown below. Each entry mode binding (e.g. x86 real mode, OS/2, Win 16 and Win32) for Socket Services has a separate section to illustrate the details of the packet for each. Where there are differences, those are highlighted with shading.

Offset	Size	Description and Usage (RE: x86 register name)
0	2	Segment or Selector of *Buffer* (ES)
2	2	Segment or Selector of *Data Pointer* (DS)
4	4	Offset of *Buffer* ([E]DI)
8	4	Offset of *Data Pointer* ([E]SI)
12	4	Reserved (BP)
16	4	Reserved (SP)
20	1	*Page* or *Socket* (BL)
21	1	*Window* (BH)
22	2	Reserved (hi word EBX)
24	4	*Attributes* ([E]DX)
28	4	*Count* ([E]CX)
32	1	*Adapter* (AL)
33	1	Service Code and RETCODE (AH)
34	2	Reserved (hi word of EAX)
36	2	Reserved (IP)
38	2	Reserved (CS)
40	2	Status - bit 0 only, all others reserved (flags)
42	2	n = Additional Arguments Buffer Length
44	n	Additional Arguments Buffer

The additional arguments are formatted where there is a control word before the data for the argument and bit 0 of that control word is a 'valid/supported' flag. This bit is set by Socket Services to inform Card Services whether or not the feature is supported and that the data in the rest of the argument is valid.

For example, lets imagine an additional argument named *Foo* that has two bytes of data. We'll use offset of x for the start of this argument. Note that the actual purpose and data contents of *1* would described in the functional portion of the Socket Services standard:

Offset	Size	Description and Usage
…	…	…
x	2	Control byte for *Foo*
		Bit 0 = Supported/Valid
		Bits 1-15 = Reserved
x+2	2	*Foo*

Please note that these are guidelines used to develop the service interfaces and exceptions have been made for specific services. See the individual service bindings for the packet interface.

Whenever possible, the interface preserves the contents of all packet fields unless they are used to return information. For bit-mapped fields, bits within a field are numbered beginning with zero. Bit 0 is the least significant bit within the field.

For all services, bit 0 of the Status field (offset 40) indicates whether the service was successful. If this bit is reset on exit, the service was successful. If this bit is set on exit, the service failed. The Return Code field always contains a RETCODE on exit.

9.6.5.2.2 Packet Interface - real-mode x86

This packet is used when the Socket Services is running in x86 real mode. In this situation the Buffer and Data Pointer arguments need x86 segments.

Offset	Size	Description and Usage
0	2	Segment of *Buffer*
2	2	Segment of *Data Pointer*
4	4	Offset of *Buffer*
8	4	Offset of *Data Pointer*
12	4	Reserved
16	4	Reserved
20	1	*Page* or *Socket*
21	1	*Window*
22	2	Reserved
24	4	*Attributes*
28	4	*Count*
32	1	*Adapter*
33	1	Service Code
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	n = Additional Arguments Buffer Length
44	n	Additional Arguments Buffer

9.6.5.2.3 Packet Interface - OS/2

This packet is used when the Socket Services is running in OS/2 protected mode. In this situation the Buffer and Data Pointer arguments need x86 selectors.

Offset	Size	Description and Usage
0	2	Selector of *Buffer*
2	2	Selector of *Data Pointer*
4	4	Offset of *Buffer*
8	4	Offset of *Data Pointer*
12	4	Reserved
16	4	Reserved
20	1	*Page* or *Socket*
21	1	*Window*
22	2	Reserved
24	4	*Attributes*
28	4	*Count*
32	1	*Adapter*
33	1	Service Code
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	n = Additional Arguments Buffer Length
44	n	Additional Arguments Buffer

9.6.5.2.4 Packet Interface - Win-16

This packet is used when the Socket Services is running in Windows 16-bit protected mode. In this situation the Buffer and Data Pointer arguments need x86 selectors.

Offset	Size	Description and Usage
0	2	Selector of *Buffer*
2	2	Selector of *Data Pointer*
4	4	Offset of *Buffer*
8	4	Offset of *Data Pointer*
12	4	Reserved
16	4	Reserved
20	1	*Page* or *Socket*
21	1	*Window*
22	2	Reserved
24	4	*Attributes*
28	4	*Count*
32	1	*Adapter*
33	1	Service Code
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	n = Additional Arguments Buffer Length
44	n	Additional Arguments Buffer

9.6.5.2.5 Packet Interface - Win32 VxD

This packet is used when the Socket Services is running as a Windows protected mode VxD. In this situation the Buffer and Data Pointer arguments are simple 32-bit pointers.

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Offset of *Buffer*
8	4	Offset of *Data Pointer*
12	4	Reserved
16	4	Reserved
20	1	*Page* or *Socket*
21	1	*Window*
22	2	Reserved
24	4	*Attributes*
28	4	*Count*
32	1	*Adapter*
33	1	Service Code
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	n = Additional Arguments Buffer Length
44	n	Additional Arguments Buffer

9.6.6 Assumptions and Constraints

This section describes assumptions and constraints of the x86 architecture binding.

9.6.6.1 ROM BIOS Located

The Socket Services interface is intended to allow the handler to be located within an IBM-PC compatible ROM BIOS. However, if Socket Services is not required for performing Initial Program Load (IPL) or bootstrap loading, Socket Services may be implemented as a device driver or a Terminate and Stay Resident (TSR) program.

9.6.6.2 Adapters Supported

The Socket Services interface allows multiple adapters containing one or more PC Card sockets. Since the *TotalAdaptors* is passed in the eight-bit [AL] register the theoretical limit is two hundred and fifty-five (255) adapters. However, the constraints imposed by locating Socket Services in ROM BIOS may impose a smaller limit. The actual limit is vendor specific.

Adapters are numbered from zero (0) to the maximum (one less than the number of adapters installed).

9.6.6.3 EDC Generators

Error Detection Code generators are optional. EDC generators are numbered from zero (0) to the maximum (one less than the number returned by **InquireAdapter**).

9.6.6.4 Sockets Supported

The Socket Services interface allows multiple PC Card sockets per adapter. The socket number is passed in the eight-bit [BL] register. However, due to the fact a bit-map of assignable sockets is used in **InquireWindow** and in **InquireEDC**, the theoretical maximum is sixteen (16) sockets per adapter. As with adapters, the constraints imposed by locating Socket Services in ROM BIOS may impose a smaller limit on the number of sockets supported. An adapter may support any number of sockets, from one to the theoretical maximum of sixteen. If a system has more than one adapter, each adapter may support a different number of sockets.

Sockets are numbered from zero (0) to one less than the number installed with a maximum of sixteen sockets per adapter.

9.6.6.5 Windows Supported

The Socket Services interface is designed without any assumptions about how or whether PC Cards are mapped into the host system's I/O or memory address space. This requires a mechanism to indicate which windows can be mapped to a particular socket. Since the number of sockets per adapter is limited to sixteen (16), the 16-bit [CX] register is used to indicate which sockets may be mapped with a particular window.

Windows are numbered from zero (0) to the maximum (one less than the number available on the adapter). Since the number of windows is returned in a byte-wide register, the theoretical maximum number is two hundred and fifty-five (255). However, since windows are identified starting with zero (0), the maximum window identifier is two hundred and fifty-four (254).

9.6.7 Individual Service Bindings

9.6.7.1 CPU Register Usage Bindings

The following sections describe how the individual services are bound when using the CPU register binding for IBM-PC compatible architectures.

9.6.7.1.1 AccessConfigurationSpace

Entry:

[AH]	ACCESS_CFG_SPACE	
[AL]	*Adapter*	
[BH]	*Function*	0..7
[BL]	*Socket*	
[CH]	*Action*	Read = 00H
		Write = 01H
[CL]	*Location*	On a four byte boundary
[EDX]	*Data*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[EDX]	*Data*	

9.6.7.1.2 AcknowledgeInterrupt

Entry:

[AH]	ACK_INTERRUPT	
[AL]	*Adapter*	0 .. Max_Adapter

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[CX]	*Sockets*	

9.6.7.1.3 GetAccessOffsets

Entry:

[AH]	ACCESS_OFFSETS	
[AL]	*Adapter*	
[BH]	*Mode*	00 = Real Mode
		01 = 16:16 Protect
		02 = 16:32 Protect
		03 = 00:32 Protect
[CX]	*NumDesired*	
[ES]:[(E)DI]	*pBuffer*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[DX]	*NumAvail*	
[ES]:[(E)DI]	*pBuffer*	

All modes return 16-bit offsets. These offsets need to be combined with information returned by **GetSSAddr** describing the location of the code segment. Offsets returned by this service are relative to the code segment.

For real-mode, 16:16 and 16:32, the routines at these offsets use FAR RET instructions to return to the caller requiring they be invoked with a FAR CALL instruction. In 0:32 (flat) protect-mode, the routines at the returned offsets use NEAR RET instructions and need to be invoked with a NEAR CALL instruction.

9.6.7.1.4 GetAdapter

Entry:

[AH]	GET_ADAPTER	
[AL]	*Adapter*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[DH]	*State*	Bit 0 = AS_POWERDOWN
		Bit 1 = AS_MAINTAIN
[DI]	*SCRouting*	Bit 0··4 = IRQ level
		Bit 6 = IRQ_HIGH
		Bit 7 = IRQ_ENABLED

9.6.7.1.5 GetAdapterCount

Entry:

[AH]	GET_ADP_CNT	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[AL]	*TotalAdapters*	If [CF] reset
		[AH] is SUCCESS and *Signature* is '**SS**'
[CX]	*Signature*	

Note: This service is used to determine if a Socket Services handler is installed in the host system. The handler may share the Socket Services interrupt vector with other, unrelated handlers. There is no guarantee these other, unrelated handlers will properly reject a Socket Services **GetAdapterCount** request. The client should confirm ***Signature*** contains '**SS**' before using ***TotalAdapters***. It is suggested the client set ***Signature*** to a value other than '**SS**' before invoking this service to insure the return value is from Socket Services and not just left over in the register from prior client activity.

9.6.7.1.6 GetBridgeWindow

Entry:

[AH]	GET_BWINDOW	
[AL]	*Adapter*	
[BH]	*Window*	

Exit:

[CF]	*Status*	set = error	
		reset = success	
[AH]	**RETCODE**		
[BL]	*Socket*		
[ECX]	*Size*	(Bytes)	
[DH]	*State*	Bit 0	WS_IO
		Bit 1	WS_ENABLED
		Bit 3	WS_PREFETCH
		Bits 3&4	WS_CACHABLE
		Bits 2, 5..7	Reserved (reset to zero)
[EDI]	*Base*	(Bytes)	

Note: If a bridge window is cachable, it is by definition prefetchable. For that reason, cachable bridge windows return both Bits 3 and 4 of the *State* field set to one.

9.6.7.1.7 GetEDC

Entry:

[AH]	GET_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[BL]	*Socket*	
[DH]	*State*	Bit 0 = EC_UNI
		Bit 1 = EC_WRITE
[DL]	*Type*	Bit 0 = ET_CHECK8
		Bit 1 = ET_SDLC16
		Bit 2 = ET_SDLC32

9.6.7.1.8 GetPage

Entry:

[AH]	GET_PAGE	
[AL]	*Adapter*	
[BH]	*Window*	
[BL]	*Page*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[DL]	*State*	Bit 0 = PS_ATTRIBUTE
		Bit 1 = PS_ENABLED
		Bit 2 = PS_WP
[DI]	*Offset*	(4 KByte units)

9.6.7.1.9 GetSetPriorHandler

Entry:

[AH]	PRIOR_HANDLER	
[AL]	*Adapter*	
[BL]	*Mode*	0 = **Get**
		1 = **Set**
[CX]:[DX]	*pHandler*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[CX]:[DX]	*pHandler*	

If this Socket Services handler is the first installed in the INT 1AH chain, the values returned by a **Get** request should be the entry point of the Time of Day handler.

One reason a **SetPriorHandler** request would fail is the Socket Services it is addressing is in ROM BIOS as the first extension to the Time of Day handler. In this case, the vector to the Time of Day handler is probably hard-coded into the ROM BIOS and not in RAM prohibiting it from being updated. This should not cause any difficulty to a client wishing to revise the chain, since this Socket Services may be bypassed by registering the values returned from a **GetPriorHandler** request to this Socket Services with a replacement Socket Services implementation.

9.6.7.1.10 GetSetSSAddr

Entry:

[AH]	SS_ADDR	
[AL]	*Adapter*	
[BH]	*Mode*	00 = Real Mode
		01 = 16:16 Protect
		02 = 16:32 Protect
		03 = 00:32 Protect
[BL]	*Subfunc*	
[CX]	*NumAddData*	
[ES]:[(E)DI]	*pBuffer*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[CX]	*NumAddData*	
[ES]:[(E)DI]	*pBuffer*	

The entry points returned by this service must receive control from a CALL instruction. The real, 16:16 and 16:32 entry points require a FAR CALL instruction to be used. The 00:32 entry point requires a NEAR CALL. When using an entry point returned by this service for any mode other than real, the client must establish a pointer to the main data area in [DS]:[(E)SI].

Note: *Subfunc* 02 is invalid, if the desired processor mode is 00 indicating real-mode.

WARNING:

Any [CS] selector created should be readable in addition to being executable to allow a Socket Services implementation to reference constant data which may reside in a ROM-ed code segment. The client must also insure that Socket Services has the appropriate privileges to allow I/O port access.

Mode specific comments have been added to the buffer entry descriptions in the tables below:

When *Subfunc* is zero (0):

Offset	Size	Description
00H	Double Word	32-bit linear base address of code segment in system memory
04H	Double Word	Limit of code segment—*Must be less than 64K in real and 16:16 protect-mode*
08H	Double Word	Entry point offset—*Must be less than 64K in real and 16:16 protect-mode*
0CH	Double Word	32-bit linear base address of main data segment in system memory—*Ignored for 0:32 (flat) protect-mode*
10H	Double Word	Limit of data segment—*Must be less than 64K in real and 16:16 protect-mode*
14H	Double Word	Data area offset—*Only used for 32-bit protect-modes*

When *Subfunc* is one (1):

Offset	Size	Description
00H	Double Word	32-bit linear base address of additional data segment—Ignored for 0:32 (flat) protect-mode
04H	Double Word	Limit of data segment—Must be less than 64K in real and 16:16 protect-mode
08H	Double Word	Data area offset—Only used for 32-bit protect-modes

When *Subfunc* is two (2):

Offset	Size	Description
00H	Double Word	32-bit offset—Ignored for 16:16 protect-modes (which assumes zero)
04H	Double Word	Selector—Ignored for 0:32 (flat) protect-mode
08H	Double Word	Reserved

When *Subfunc* is four (4):

Offset	Size	Description
00H	Double Word	32-bit linear base address of code segment in system memory
04H	Double Word	Limit of code segment—*Must be less than 64K in real and 16:16 protect-mode*
08H	Double Word	Entry point offset (entry point that utilizes the packet interface)—*Must be less than 64K in real and 16:16 protect-mode*
0CH	Double Word	32-bit linear base address of main data segment in system memory—*Ignored for 0:32 (flat) protect-mode*
10H	Double Word	Limit of data segment—*Must be less than 64K in real and 16:16 protect-mode*
14H	Double Word	Data area offset—*Only used for 32-bit protect-modes*

9.6.7.1.11 GetSocket

Entry:

[AH]	GET_SOCKET	
[AL]	*Adapter*	
[BL]	*Socket*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[BH]	*SCIntMask*	(Uses the same bit masks as **InquireSocket**)
[CH]		Lower Nibble = *VccLevel*
		Upper Nibble = *Vcontrol* Bit 4 = VCTL_CISREAD Bit 5 = VCTL_OVERRIDE Bit 6··7 = Voltage Sense Signaling (read-only) 0 = VCTL_5V 1 = VCTL_33V 2 = VCTL_XXV 3 = Reserved (not used)
[CL]	*VppLevels*	Lower Nibble = **VPP2**
		Upper Nibble = **VPP1**
[DH]	*State*	(Uses same bit masks as *SCIntMask*)
[DL]	*CtlInd*	(Uses same bit masks as **InquireSocket**)
[DI]		Low Byte = *IREQRouting*
		High Byte = *IFType* Bit 0··4 IRQ level Bit 5 RESERVED Bit 6 IRQ_HIGH Bit 7 IRQ_ENABLE Bit 8··9 Interface type 0 = IF_CARDBUS, 1 = IF_MEMORY 2 = IF_IO 3 = IF_CUSTOM Bits 10··11 DREQ 0 = No DMA 1 = **SPKR#** 2 = **IOIS16#** 3 = **INPACK#** Bits 12··15 DMA Channel (0··15)
[BP]	*IFIndex*	Index of custom interface when IFType = IF_CUSTOM

9.6.7.1.12 GetSSInfo

Entry:

[AH]	GET_SS_INFO	
[AL]	*Adapter*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[AL]	Zero (0)	Insures backward compatibility with Release 1.01
[BX]	*Compliance*	
[CH]	*NumAdapters*	
[CL]	*FirstAdapter*	

9.6.7.1.13 GetStatus

Entry:

[AH]	GET_STATUS	
[AL]	*Adapter*	
[BL]	*Socket*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[BH]	*CardState*	(same bit-masks as **GetSocket** *SCIntMask*)
[DH]	*SocketState*	(same as **GetSocket**)
[DL]	*CtlInd*	(same as **GetSocket**)
[DI]		High Byte = *IFType*
		Low Byte = *IREQRouting*
		(same as **GetSocket**)

9.6.7.1.14 GetVendorInfo

Entry:

[AH]	GET_VENDOR_INFO	
[AL]	*Adapter*	
[BL]	*Type*	
[ES]:[(E)DI]	*pBuffer*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[ES]:[(E)DI]	*pBuffer*	
[DX]	*Release*	

9.6.7.1.15 GetWindow

Entry:

[AH]	GET_WINDOW	
[AL]	*Adapter*	
[BH]	*Window*	

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[BL]	*Socket*	and parameter width Bit 0··3 = Socket Number Bit 4 = *Size* and *Base* width 0 = 16 bits 1 = 32 bits
[(E)CX]	*Size*	If bit 4 of [BL] is reset, I/O windows are expressed in bytes, memory windows are expressed in 4 KByte units and [CX] is used. If bit 4 of [BL] is set, both I/O and memory windows are expressed in bytes and [ECX] is used.
[DH]	*State*	Bit 0 = WS_IO Bit 1 = WS_ENABLED Bit 2 = WS_16BIT Bit 3 = WS_PAGED (Memory window) or WS_EISA (I/O window) Bit 4 = WS_CENABLE (I/O window with WS_EISA set)
[DL]	*Speed*	
[(E)DI]	*Base*	If bit 4 of [BL] is reset, I/O windows are expressed in bytes, memory windows are expressed in 4 KByte units and [DI] is used. If bit 4 of [BL] is set, both I/O and memory windows are expressed in bytes and [EDI] is used.

9.6.7.1.16 InquireAdapter

Entry:

[AH]	INQ_ADAPTER	
[AL]	*Adapter*	
[ES]:[(E)DI]	*pBuffer*	for adapter characteristics and power levels

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[BH]	*NumWindows*	
[BL]	*NumSockets*	
[CX]	*NumEDCs*	
[DX]	*NumBridgeWindows*	
[ES]:[(E)DI]	*pBuffer*	with adapter characteristics and power management tables

9.6.7.1.17 InquireBridgeWindow

Entry:

[AH]	INQ_BWINDOW	
[AL]	*Adapter*	
[BH]	*Window*	
[ES]:[(E)DI]	*pBuffer*	for window characteristics

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[BL]	*WndCaps*	Bit 0 = WC_MEMORY Bit 1 = Reserved (reset to zero) Bit 2 = WC_IO Bit 3··7 = Reserved (reset to zero)
[CX]	*Sockets*	Bit 0··15 = Bit-mask Bit 0 is Socket 0 Bit 1 is Socket 1 etc.
[ES]:[(E)DI]	*pBuffer*	with window characteristics

Note: All BASE and SIZE values in the BIOWINTBL and BMEMWINTBL structures returned by this service are 32-bits wide. That is, the BASE32 and WSIZE32 data types are used for BASE and SIZE values.

9.6.7.1.18 InquireEDC

Entry:

[AH]	INQ_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[CX]	*Sockets*	Bit 0··15 = Bit-mask Bit 0 is Socket 0 Bit 1 is Socket 1 etc.
[DH]	*Caps*	Bit 0 = EC_UNI Bit 1 = EC_BI Bit 2 = EC_REGISTER Bit 3 = EC_MEMORY Bit 4 = EC_PAUSABLE
[DL]	*Types*	Bit 0 = ET_CHECK8 Bit 1 = ET_SDLC16 Bit 2 = ET_SDLC32

9.6.7.1.19 InquireSocket

Entry:

[AH]	INQ_SOCKET	
[AL]	*Adapter*	
[BL]	*Socket*	
[ES]:[(E)DI]	*pBuffer*	for socket characteristics

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[BH]	*SCIntCaps*	Bit 0 = SBM_WP Bit 1 = SBM_LOCKED Bit 2 = SBM_EJECT Bit 3 = SBM_INSERT Bit 4 = SBM_BVD1 Bit 5 = SBM_BVD2 Bit 6 = SBM_RDYBSY Bit 7 = SBM_CD
[DH]	*SCRptCaps*	(same as *SCIntCaps*)
[DL]	*CtlIndCaps*	Bit 0 = SBM_WP Bit 1 = SBM_LOCKED Bit 2 = SBM_EJECT Bit 3 = SBM_INSERT Bit 4 = SBM_LOCK Bit 5 = SBM_BATT Bit 6 = SBM_BUSY Bit 7 = SBM_XIP
[ES]:[(E)DI]	*pBuffer*	with socket characteristics

9.6.7.1.20 InquireWindow

Entry:

[AH]	INQ_WINDOW	
[AL]	*Adapter*	
[BH]	*Window*	if [BH] is 0FFH then Window is passed in [DH] and the window characteristics returned in the Buffer use 32-bit wide values for BASE and SIZE
[DH]	*Window*	if [BH] is 0FFH, otherwise undefined
[ES]:[(E)DI]	*pBuffer*	for window characteristics

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	
[BL]	*WndCaps*	Bit 0 = WC_COMMON Bit 1 = WC_ATTRIBUTE Bit 2 = WC_IO Bit 7 = WC_WAIT
[CX]	*Sockets*	Bit 0··15 = Bit-mask Bit 0 is Socket 0 Bit 1 is Socket 1 etc.
[ES]:[(E)DI]	*pBuffer*	with window characteristics

Note: The data types used for the BASE and SIZE values in the IOWINTBL and MEMWINTBL structures returned by this service vary depending on the value passed in the [BH] register.

If [BH] is not FFH, the BASE and SIZE values are 16-bits wide using the BASE16 and WSIZE16 data types. When [BH] is not FFH, BASE and SIZE values in the IOWINTBL structure are expressed in bytes and BASE and SIZE values in the MEMWINTBL structure are expressed in 4 KByte units.

If [BH] is FFH, the BASE and SIZE values are 32-bits wide using the BASE32 and WSIZE32 data types. When [BH] is FFH, BASE and SIZE values in both the IOWINTBL and MEMWINTBL structures are expressed in bytes.

This encoding allows backward compatibility with prior releases of the Socket Services binding for this service that only used 16-bit values for BASE and SIZE.

9.6.7.1.21 PauseEDC

Entry:

[AH]	PAUSE_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.22 ReadEDC

Entry:

[AH]	READ_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	
[DX]	*Value*	

9.6.7.1.23 ResetSocket

Entry:

[AH]	RESET_SOCKET	
[AL]	*Adapter*	
[BL]	*Socket*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.24 ResumeEDC

Entry:

[AH]	RESUME_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.25 SetAdapter

Entry:

[AH]	SET_ADAPTER	
[AL]	*Adapter*	
[DH]	*State*	(same as **GetAdapter**)
[DI]	*SCRouting*	(same as **GetAdapter**)

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.26 SetBridgeWindow

Entry:

[AH]	GET_BWINDOW	
[AL]	*Adapter*	
[BH]	*Window*	
[BL]	*Socket*	
[ECX]	*Size*	(Bytes)
[DH]	*State*	(Same as **GetBridgeWindow)**
[EDI]	*Base*	(Bytes)

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.27 SetEDC

Entry:

[AH]	SET_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	
[BL]	*Socket*	
[DH]	*State*	(same as **GetEDC**)
[DL]	*Type*	(same as **GetEDC**)

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.28 SetPage

Entry:

[AH]	SET_PAGE	
[AL]	*Adapter*	
[BH]	*Window*	
[BL]	*Page*	
[DH]	*State*	(same as **GetPage**)
[DI]	*Offset*	(4 KByte units)

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.29 SetSocket

Entry:

[AH]	SET_SOCKET	
[AL]	*Adapter*	
[BL]	*Socket*	
[BH]	*SCIntMask*	(same as **GetSocket**)
[CH]	*Vcontrol*	(same as **GetSocket**)
[CL]	*VppLevels*	(same as **GetSocket**
[DH]	*State*	(same as **GetSocket**)
[DL]	*CtlInd*	(same as **GetSocket**)
[DI]		High Byte = *IFType* Low Byte = *IREQRouting* (same as **GetSocket**), plus: Bit 5 IRQ_INVALID
[BP]	*IFIndex*	Index of custom interface when IFType = IF_CUSTOM (same as **GetSocket**)

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	

9.6.7.1.30 SetWindow

Entry:

[AH]	SET_WINDOW	
[AL]	*Adapter*	
[BH]	*Window*	
[BL]	*Socket*	
[CX]	*Size*	(same as **GetWindow**)
[DH]	*State*	(same as **GetWindow**)
[DL]	*Speed*	
[DI]	*Base*	(same as **GetWindow**)

Exit:

[CF]	*Status*	set = error reset = success
[AH]	**RETCODE**	

9.6.7.1.31 StartEDC

Entry:

[AH]	START_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.32 StopEDC

Entry:

[AH]	STOP_EDC	
[AL]	*Adapter*	
[BH]	*EDC*	

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.1.33 VendorSpecific

Entry:

[AH]	VEND_SPECIFIC	
[AL]	*Adapter*	

All other registers are vendor specific.

Exit:

[CF]	*Status*	set = error
		reset = success
[AH]	**RETCODE**	

9.6.7.2 Packet Usage Bindings

The following sections describe how the individual services are bound when using the packet binding. Specifically, the packets are used as described in ***9.6.5.2 Packet Interface Usage*** and these sections describe only extensions or differences. When a parameter or field is different for entry and exit then the syntax of 'entry / exit' is used for differentiation in the desciption.

9.6.7.2.1 AccessConfigurationSpace

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	*Function* (0..7 only)
22	2	Reserved
24	4	*Data*
28	1	*Location* (on a four byte boundary)
29	1	Action (Read = 00h, Write = 01h)
30	2	Reserved
32	1	*Adapter*
33	1	ACCESS_CFG_SPACE and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.2 AcknowledgeInterrupt

Offset	Size	Description and Usage
0	2	Segment or Selector of *Buffer*
2	2	Segment or Selector of *Data Pointer*
4	4	Offset of *Buffer*
8	4	Offset of *Data Pointer*
12	4	Reserved
16	4	Reserved
20	1	*Page* or *Socket*
21	1	*Window*
22	2	Reserved
24	4	*Attributes*
28	4	Reserved / *Sockets*
32	1	*Adapter*
33	1	ACK_INTERRUPT and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.3 GetAccessOffsets

Offset	Size	Description and Usage (RE: x86 register name)
0	2	Segment or Selector of *pBuffer* (ES)
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*Mode* 00 = Real Mode 01 = 16:16 Protect 02 = 16:32 Protect 03 = 00:32 Protect
22	2	Reserved
24	4	Reserved
28	4	*NumDesired*
32	1	*Adapter*
33	1	ACCESS_OFFSETS and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

All modes return 16-bit offsets. These offsets need to be combined with information returned by **GetSSAddr** describing the location of the code segment. Offsets returned by this service are relative to the code segment.

For real-mode, 16:16 and 16:32, the routines at these offsets use FAR RET instructions to return to the caller requiring they be invoked with a FAR CALL instruction. In 0:32 (flat) protect-mode, the routines at the returned offsets use NEAR RET instructions and need to be invoked with a NEAR CALL instruction.

9.6.7.2.4 GetAdapter

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved / *SCRouting* Bit 0··4 = IRQ level
		Bit 6 = IRQ_HIGH
		Bit 7 = IRQ_ENABLED
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	Reserved
22	2	Reserved
24	1	Reserved
25	1	Reserved / *State*: Bit 0 = AS_POWERDOWN
		Bit 1 = AS_MAINTAIN
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	GET_ADAPTER and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.5 GetAdapterCount

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	Reserved
22	2	Reserved
24	4	Reserved
28	4	Reserved / *Signature*
32	1	Reserved / *TotalAdapters (*if *Status* bit 0 reset then RETCODE=SUCCESS and *Signature* is 'SS')
33	1	GET_ADP_CNT and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

Note: This service is used to determine if a Socket Services handler is installed in the host system. The handler may share the Socket Services interrupt vector with other, unrelated handlers. There is no guarantee these other, unrelated handlers will properly reject a Socket Services **GetAdapterCount** request. The client should confirm ***Signature*** contains '**SS**' before using ***TotalAdapters***. It is suggested the client set ***Signature*** to a value other than '**SS**' before invoking this service to insure the return value is from Socket Services and not just left over in the register from prior client activity.

9.6.7.2.6 GetBridgeWindow

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved / *Base* (Bytes)
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *Socket*
21	1	Reserved / *Window*
22	2	Reserved
24	1	Reserved
25	1	Reserved / *State*: Bit 0 WS_IO Bit 1 WS_ENABLED Bit 3 WS_PREFETCH Bits 3&4 WS_CACHABLE Bits 2, 5..7 Reserved (reset to zero)
26	2	Reserved
28	4	Reserved / *Size*
32	1	*Adapter*
33	1	GET_BWINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

Note: If a bridge window is cachable, it is by definition prefetchable. For that reason, cachable bridge windows return both Bits 3 and 4 of the *State* field set to one.

9.6.7.2.7 GetEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *Socket*
21	1	*EDC*
22	2	Reserved
24	1	Reserved / *Type*:
		Bit 0 = ET_CHECK8
		Bit 1 = ET_SDLC16
		Bit 2 = ET_SDLC32
25	1	Reserved / *State*:
		Bit 0 = EC_UNI
		Bit 1 = EC_WRITE
24	4	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	GET_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.8 GetPage

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved / *Offset*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Page*
21	1	*Window*
22	2	Reserved
24	1	Reserved / *State*: Bit 0 = PS_ATTRIBUTE Bit 1 = PS_ENABLED Bit 2 = PS_WP
25	1	Reserved
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	GET_PAGE and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.9 GetSetPriorHandler

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Mode*:
		0 = **Get**
		1 = **Set**
21	1	Reserved
22	2	Reserved
24	8	*pHandler*
32	1	*Adapter*
33	1	PRIOR_HANDLER and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

If this Socket Services handler is the first installed in the INT 1AH chain, the values returned by a **Get** request should be the entry point of the Time of Day handler.

One reason a **SetPriorHandler** request would fail is the Socket Services it is addressing is in ROM BIOS as the first extension to the Time of Day handler. In this case, the vector to the Time of Day handler is probably hard-coded into the ROM BIOS and not in RAM prohibiting it from being updated. This should not cause any difficulty to a client wishing to revise the chain, since this Socket Services may be bypassed by registering the values returned from a **GetPriorHandler** request to this Socket Services with a replacement Socket Services implementation.

9.6.7.2.10 GetSetSSAddr

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Subfunc*
21	1	*Mode*: 00 = Real Mode 01 = 16:16 Protect 02 = 16:32 Protect 03 = 00:32 Protect
22	2	Reserved
24	4	Reserved
28	4	*NumAddData*
32	1	*Adapter*
33	1	SS_ADDR and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

The entry points returned by this service must receive control from a CALL instruction. The real, 16:16 and 16:32 entry points require a FAR CALL instruction to be used. The 00:32 entry point requires a NEAR CALL. When using an entry point returned by this service for any mode other than real, the client must establish a pointer to the main data area in offsets 2 and 8.

Note: *Subfunc* 02 is invalid, if the desired processor mode is 00 indicating real-mode.

WARNING:

Any Code Segment selector created should be readable in addition to being executable to allow a Socket Services implementation to reference constant data which may reside in a ROM-ed code segment. The client must also insure that Socket Services has the appropriate privileges to allow I/O port access.

Mode specific comments have been added to the buffer entry descriptions in the tables below:

When *Subfunc* is zero (0):

Offset	Size	Description
00H	Double Word	32-bit linear base address of code segment in system memory
04H	Double Word	Limit of code segment—*Must be less than 64K in real and 16:16 protect-mode*
08H	Double Word	Entry point offset—*Must be less than 64K in real and 16:16 protect-mode*
0CH	Double Word	32-bit linear base address of main data segment in system memory—*Ignored for 0:32 (flat) protect-mode*
10H	Double Word	Limit of data segment—*Must be less than 64K in real and 16:16 protect-mode*
14H	Double Word	Data area offset—*Only used for 32-bit protect-modes*

When *Subfunc* is one (1):

Offset	Size	Description
00H	Double Word	32-bit linear base address of additional data segment—Ignored for 0:32 (flat) protect-mode
04H	Double Word	Limit of data segment—Must be less than 64K in real and 16:16 protect-mode
08H	Double Word	Data area offset—Only used for 32-bit protect-modes

When *Subfunc* is two (2):

Offset	Size	Description
00H	Double Word	32-bit offset—Ignored for 16:16 protect-modes (which assumes zero)
04H	Double Word	Selector—Ignored for 0:32 (flat) protect-mode
08H	Double Word	Reserved

9.6.7.2.11 GetSocket

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	2	Reserved / Low Byte = *IREQRouting*
		High Byte = *IFType* Bit 0··4 IRQ level Bit 5 RESERVED Bit 6 IRQ_HIGH Bit 7 IRQ_ENABLE Bit 8··9 Interface type 0 = IF_CARDBUS, 1 = IF_MEMORY 2 = IF_IO 3 = IF_CUSTOM Bits 10··11 DREQ 0 = No DMA 1 = **SPKR#** 2 = **IOIS16#** 3 = **INPACK#** Bits 12··15 DMA Channel (0··15)
6	2	Reserved
8	4	Reserved
12	4	Reserved / *IFIndex* (Index of custom interface when IFType = IF_CUSTOM)
16	4	Reserved
20	1	*Socket*
21	1	Reserved / *SCIntMask* (Uses the same bit masks as **InquireSocket**)
22	2	Reserved
24	1	Reserved / *CtlInd* (Uses same bit masks as **InquireSocket**)
25	1	Reserved / *State* (Uses same bit masks as **InquireSocket**)
26	2	Reserved
28	1	Reserved / Vpp Levels: Lower Nibble = **VPP2** Upper Nibble = **VPP1**
29	1	Reserved / Lower Nibble = *VccLevel* Upper Nibble = *Vcontrol* Bit 4 = VCTL_CISREAD Bit 5 = VCTL_OVERRIDE Bit 6··7 = Voltage Sense Signaling (read-only) 0 = VCTL_5V 1 = VCTL_33V 2 = VCTL_XXV 3 = Reserved (not used)
30	2	Reserved
32	1	*Adapter*
33	1	GET_SOCKET and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.12 GetSSInfo

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	2	Reserved / *Compliance*
22	2	Reserved
24	4	Reserved
28	1	Reserved / *FirstAdapter*
29	1	Reserved / *NumAdapters*
30	2	Reserved
32	1	*Adapter* / Zero (0) (will be zero on return for compatibility)
33	1	GET_SS_INFO and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.13 GetStatus

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	2	Reserved / Interface and IREQ Routing High Byte = *IFType* Low Byte = *IREQRouting* (same as **GetSocket**)
6	2	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	Reserved / *CardState* (same bit mask as **GetSocket** *SCIntMask*)
22	2	Reserved
24	1	Reserved / *CtlInd* (Same as GetSocket)
25	1	Reserved / *SocketState* (same as **GetSocket**)
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	GET_STATUS and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.14 GetVendorInfo

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Type*
21	1	Reserved
22	2	Reserved
24	2	Reserved / *Release*
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	GET_VENDOR_INFO and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.15 GetWindow

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved / *Base*: If bit 4 of the *Socket* field (offset 20) is reset, I/O windows are expressed in bytes, memory windows are expressed in 4 Kbyte units and only the low 16-bits are used for *Base*. If bit 4 of the *Socket* field (offset 20) is set, both I/O and memory windows are expressed in bytes and all 32-bits of *Base* is used.
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *Socket* and parameter width Bit 0··3 = Socket Number Bit 4 = *Size* and *Base* width 0 = 16 bits 1 = 32 bits
21	1	*Window*
22	2	Reserved
24	1	Reserved / *Speed*
25	1	Reserved / *State*: Bit 0 = WS_IO Bit 1 = WS_ENABLED Bit 2 = WS_16BIT Bit 3 = WS_PAGED (Memory window) or WS_EISA (I/O window) Bit 4 = WS_CENABLE (I/O window with WS_EISA set)
24	2	Reserved
28	4	Reserved / *Size*: If bit 4 of the *Socket* field (offset 20) is reset, I/O windows are expressed in bytes, memory windows are expressed in 4 Kbyte units and only 16 bits is used for *Size*. If bit 4 of the *Socket* field (offset 20) is set, both I/O and memory windows are expressed in bytes and 32 bits are used for *Size*.
32	1	*Adapter*
33	1	GET_WINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.16 InquireAdapter

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *NumSockets*
21	1	Reserved / *NumWindows*
22	2	Reserved
24	4	Reserved / *NumBridgeWindows*
28	4	Reserved / *NumEDCs*
32	1	*Adapter*
33	1	INQ_ADAPTER and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.17 InquireBridgeWindow

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *WndCaps*: Bit 0 = WC_MEMORY Bit 1 = Reserved (reset to zero) Bit 2 = WC_IO Bit 3··7 = Reserved (reset to zero)
21	1	*Window*
22	2	Reserved
24	4	Reserved
28	4	Reserved / *Sockets*: Bit 0··15 = Bit-mask Bit 0 is Socket 0 Bit 1 is Socket 1 etc.
32	1	*Adapter*
33	1	INQ_BWINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

Note: All BASE and SIZE values in the BIOWINTBL and BMEMWINTBL structures returned by this service are 32-bits wide. That is, the BASE32 and WSIZE32 data types are used for BASE and SIZE values.

9.6.7.2.18 InquireEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	1	Reserved / *Types*: Bit 0 = ET_CHECK8 Bit 1 = ET_SDLC16 Bit 2 = ET_SDLC32
25	1	Reserved / *Caps*: Bit 0 = EC_UNI Bit 1 = EC_BI Bit 2 = EC_REGISTER Bit 3 = EC_MEMORY Bit 4 = EC_PAUSABLE
24	4	Reserved
28	4	Reserved / *Sockets*: Bit Mask
32	1	*Adapter*
33	1	INQ_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.19 InquireSocket

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	Reserved / *SCIntCaps*: Bit 0 = SBM_WP Bit 1 = SBM_LOCKED Bit 2 = SBM_EJECT Bit 3 = SBM_INSERT Bit 4 = SBM_BVD1 Bit 5 = SBM_BVD2 Bit 6 = SBM_RDYBSY Bit 7 = SBM_CD
22	2	Reserved
24	1	Reserved / *CtlIndCaps*: Bit 0 = SBM_WP Bit 1 = SBM_LOCKED Bit 2 = SBM_EJECT Bit 3 = SBM_INSERT Bit 4 = SBM_LOCK Bit 5 = SBM_BATT Bit 6 = SBM_BUSY Bit 7 = SBM_XIP
25	1	Reserved / *SCRptCaps* (same as *SCIntCaps*)
24	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	INQ_SOCKET and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.20 InquireWindow

Offset	Size	Description and Usage
0	2	Segment or Selector of *pBuffer*
2	2	Reserved
4	4	Offset of *pBuffer*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved / *WndCaps*: Bit 0 = WC_COMMON Bit 1 = WC_ATTRIBUTE Bit 2 = WC_IO Bit 7 = WC_WAIT
21	1	*Window*: if this field value is 0FFH then *Window* is passed in offset 25 and the window characteristics returned in the Buffer use 32-bit wide values for BASE and SIZE
22	2	Reserved
24	1	Reserved
25	1	*Window* If offset 21 is 0FFH else Reserved
24	2	Reserved
28	4	Reserved / *Sockets*: Bit 0··15 = Bit-mask Bit 0 is Socket 0 Bit 1 is Socket 1 etc.
32	1	*Adapter*
33	1	INQ_WINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

Note: The data types used for the BASE and SIZE values in the IOWINTBL and MEMWINTBL structures returned by this service vary depending on the value passed in the *Window* (offset 21) field.

If the *Window* (offset 21) field is not FFH, the BASE and SIZE values are 16-bits wide using the BASE16 and WSIZE16 data types. When the *Window* (offset 21) field is not FFH, BASE and SIZE values in the IOWINTBL structure are expressed in bytes and BASE and SIZE values in the MEMWINTBL structure are expressed in 4 KByte units.

If the *Window* (offset 21) field is FFH, the BASE and SIZE values are 32-bits wide using the BASE32 and WSIZE32 data types. When the *Window* (offset 21) field is FFH, BASE and SIZE values in both the IOWINTBL and MEMWINTBL structures are expressed in bytes.

This encoding allows backward compatibility with prior releases of the Socket Services for this service that only used 16-bit values for BASE and SIZE.

9.6.7.2.21 PauseEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	4	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	PAUSE_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.22 ReadEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	4	Reserved / *Value*
28	4	Reserved
32	1	*Adapter*
33	1	READ_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.23 ResetSocket

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	Reserved
22	2	Reserved
24	4	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	RESET_SOCKET and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.24 ResumeEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	4	*Value*
28	4	Reserved
32	1	*Adapter*
33	1	RESUME_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.25 SetAdapter

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	*SCRouting* (same as **GetAdapter**)
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	Reserved
22	2	Reserved
24	1	*State* (same as in **GetAdapter**)
25	1	Reserved
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	SET_ADAPTER and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.26 SetBridgeWindow

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	*Base*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	*Window*
22	2	Reserved
24	1	State (same as in **GetBridgeWindow**)
25	1	Reserved
26	2	Reserved
28	4	*Size* (bytes)
32	1	*Adapter*
33	1	SET_BWINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.27 SetEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket*
21	1	Reserved
22	2	Reserved
24	1	*Type* (same as in **GetEDC**)
25	1	*State* (same as in **GetEDC**)
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	SET_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.28 SetPage

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	*Offset*
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Page*
21	1	*Window*
22	2	Reserved
24	1	*State*: (same as in **GetPage**)
25	1	Reserved
26	2	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	SET_PAGE and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.29 SetSocket

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	2	High Byte = *IFType* Low Byte = *IREQRouting* (same as **GetSocket**), plus: Bit 5 IRQ_INVALID
6	2	Reserved
8	4	Reserved
12	4	*IFIndex* (same as in **GetSocket**)
16	4	Reserved
20	1	*Socket*
21	1	*SCIntMask* (same as in **GetSocket**)
22	2	Reserved
24	1	*CtlInd* (same as in **GetSocket**)
25	1	*State* (same as in **GetSocket**)
26	2	Reserved
28	1	*VppLevels*: (same as in **GetSocket**)
29	1	*VControl* (same as in **GetSocket**)
30	2	Reserved
32	1	*Adapter*
33	1	SET_SOCKET and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.30 SetWindow

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	*Base* (same as in **GetWindow**)
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	*Socket* (same as in **GetWindow**)
21	1	*Window*
22	2	Reserved
24	1	*Speed*
25	1	*State* (same as in **GetWindow**)
24	2	Reserved
28	4	*Size* (same as in **GetWindow**)
32	1	*Adapter*
33	1	SET_WINDOW and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.31 StartEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	4	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	START_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.32 StopEDC

Offset	Size	Description and Usage
0	2	Reserved
2	2	Reserved
4	4	Reserved
8	4	Reserved
12	4	Reserved
16	4	Reserved
20	1	Reserved
21	1	*EDC*
22	2	Reserved
24	4	Reserved
28	4	Reserved
32	1	*Adapter*
33	1	STOP_EDC and RETCODE
34	2	Reserved
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	0 = Additional Arguments Buffer Length
44	0	Additional Arguments Buffer

9.6.7.2.33 VendorSpecific

Offset	Size	Description and Usage
0	2	Vendor Specific
2	2	Vendor Specific
4	4	Vendor Specific
8	4	Vendor Specific
12	4	Reserved
16	4	Reserved
20	1	Vendor Specific
21	1	Vendor Specific
22	2	Vendor Specific
24	4	Vendor Specific
28	4	Vendor Specific
32	1	*Adapter*
33	1	VEND_SPECIFIC and RETCODE
34	2	Vendor Specific
36	2	Reserved
38	2	Reserved
40	2	Status - bit 0 only, all others reserved
42	2	n = Additional Arguments Buffer Length (vendor specific)
44	n	Additional Arguments Buffer - Vendor Specific

9.6.8 Assembly Language Definitions

This section contains suggested assembly language definitions for values required by clients of the Socket Services interface.

```
; The following definitions are formatted for Microsoft MASM 6.0.

; ----- Service definitions

GET_ADP_CNT     EQU   80H
                      ; 81H and 82H reserved for historical purposes
GET_SS_INFO     EQU   83H
INQ_ADAPTER     EQU   84H
GET_ADAPTER     EQU   85H
SET_ADAPTER     EQU   86H
INQ_WINDOW      EQU   87H
GET_WINDOW      EQU   88H
SET_WINDOW      EQU   89H
GET_PAGE        EQU   8AH
SET_PAGE        EQU   8BH
INQ_SOCKET      EQU   8CH
GET_SOCKET      EQU   8DH
SET_SOCKET      EQU   8EH
GET_STATUS      EQU   8FH
RESET_SOCKET    EQU   90H
                      ; 91H thru 94H reserved for historical purposes
INQ_EDC         EQU   95H
GET_EDC         EQU   96H
SET_EDC         EQU   97H
START_EDC       EQU   98H
PAUSE_EDC       EQU   99H
RESUME_EDC      EQU   9AH
STOP_EDC        EQU   9BH
READ_EDC        EQU   9CH
GET_VENDOR_INFO EQU   9DH
ACK_INTERRUPT   EQU   9EH
PRIOR_HANDLER   EQU   9FH
SS_ADDR         EQU   0A0H
ACCESS_OFFSETS  EQU   0A1H
ACCESS_CONFIG   EQU   0A2H
INQ_BWINDOW     EQU   0A3H
GET_BWINDOW     EQU   0A4H
SET_BWINDOW     EQU   0A5H
                      ; A6H thru ADH are reserved for expansion
VEND_SPECIFIC   EQU   0AEH
CARD_SERVICES   EQU   0AFH
SS_INT          EQU   1AH    ; Socket Services Int vector
```

```
; ----- Return codes

SUCCESS         EQU   00H
BAD_ADAPTER     EQU   01H
BAD_ATTRIBUTE   EQU   02H
BAD_BASE        EQU   03H
BAD_EDC         EQU   04H
                      ; 05H reserved for historical purposes
BAD_IRQ         EQU   06H
BAD_OFFSET      EQU   07H
BAD_PAGE        EQU   08H
READ_FAILURE    EQU   09H
BAD_SIZE        EQU   0AH
BAD_SOCKET      EQU   0BH
                      ; 0CH is reserved for historical purposes
BAD_TYPE        EQU   0DH
BAD_VCC         EQU   0EH
BAD_VPP         EQU   0FH
                      ; 10H is reserved for historical purposes
BAD_WINDOW      EQU   11H
WRITE_FAILURE   EQU   12H
                      ; 13H is reserved for historical purposes
NO_CARD         EQU   14H
BAD_SERVICE     EQU   15H
BAD_MODE        EQU   16H
BAD_SPEED       EQU   17H
BUSY            EQU   18H

; ----- Defined data types

ADAPTER  TYPEDEF     BYTE
BASE16   TYPEDEF     WORD
BASE32   TYPEDEF     DWORD
BCD      TYPEDEF     WORD
COUNT    TYPEDEF     BYTE
EDC      TYPEDEF     BYTE
FLAGS8   TYPEDEF     BYTE
FLAGS16  TYPEDEF     WORD
FLAGS32  TYPEDEF     DWORD
IRQ      TYPEDEF     BYTE
COFFSET  TYPEDEF     WORD  ;     OFFSET is reserved by MASM 6.0
WPAGE    TYPEDEF     BYTE  ;     PAGE is reserved by MASM 6.0
PWRINDEX TYPEDEF     BYTE
RETCODE  TYPEDEF     BYTE
SIGNATURE TYPEDEF    WORD
WSIZE16  TYPEDEF     WORD  ;     SIZE is reserved by MASM 6.0
WSIZE32  TYPEDEF     DWORD
SOCKET   TYPEDEF     BYTE
SPEED    TYPEDEF     BYTE
WINDOW   TYPEDEF     BYTE
SKTBITS  TYPEDEF     WORD
```

```
; ----- Structures

PWRENTRY STRUCT             ;     Power level and valid signals
PowerLevel     PWRINDEX     ?     ;     as returned by InquireAdapter
ValidSignals   FLAGS8       ?
PWRENTRY ENDS

ACHARTBL       STRUCT             ;     Inquire Adapter
AdpCaps  FLAGS16     ?
ActiveHi FLAGS32     ?
ActiveLo FLAGS32     ?
ACHARTBL ENDS


SCHARTBL       STRUCT             ;     Inquire Socket
SktCaps  FLAGS16     ?
ActiveHi FLAGS32     ?
ActiveLo FLAGS32     ?
DMAChannels    FLAGS16     ?
wNumCustomIF   WORD  ?
dCustomIF      DWORD ?
CHARTBL  ENDS


MEMWINTBL      STRUCT             ;     Inquire Window for Memory Windows
MemWndCaps     FLAGS16     ?      ;     Window Capabilities Flags
FirstByte      BASE        ?      ;     System Address of First Byte
LastByte       BASE        ?      ;     System Address of Last Byte
MinSize        WSIZE       ?      ;     Minimum Window Size
MaxSize        WSIZE       ?      ;     Maximum Window Size
ReqGran        WSIZE       ?      ;     Size Granularity
ReqBase        WSIZE       ?      ;     Window Base Alignment requirement
ReqOffset      WSIZE       ?      ;     Alignment Requirement for offsets
Slowest        SPEED       ?      ;     Slowest Access Speed for Window
Fastest        SPEED       ?      ;     Fastest Access Speed for Window
MEMWINTBL      ENDS

IOWINTBL STRUCT             ;     Inquire Window for IO Windows
IOWndCaps      FLAGS16     ?      ;     Window Capabilities Flags
FirstByte      BASE        ?      ;     System Address of First Byte
LastByte       BASE        ?      ;     System Address of Last Byte
MinSize        WSIZE       ?      ;     Minimum Window Size
MaxSize        WSIZE       ?      ;     Maximum Window Size
ReqGran        WSIZE       ?      ;     Size Granularity
AddrLines      COUNT       ?      ;     Address Lines Decoded by Window
EISASlot       FLAGS8      ?      ;     Upper 4 I/O Address lines for EISA
IOWINTBL ENDS

; ----- Valid power level bit-masks

VCC   EQU      10000000B
VPP1  EQU      01000000B
VPP2  EQU      00100000B
```

```
; ----- Adapter capabilities bit-masks

AC_IND          EQU   0000000000000001B
AC_PWR          EQU   0000000000000010B
AC_DBW          EQU   0000000000000100B
AC_CARDBUS      EQU   0000000000001000B

; ----- Adapter state

AS_POWERDOWN    EQU   00000001B
AS_MAINTAIN     EQU   00000010B

; ----- Generic window capabilities bit-masks

WC_COMMON       EQU   00000001B
WC_ATTRIBUTE    EQU   00000010B
WC_IO           EQU   00000100B
WC_WAIT         EQU   10000000B

; ----- Generic bridge window capabilities bit-masks

WC_MEMORY       EQU   00000001B

; ----- Bridge, Memory and I/O window capabilities bit-masks

WC_BASE         EQU   0000000000000001B
WC_SIZE         EQU   0000000000000010B
WC_WENABLE      EQU   0000000000000100B
WC_8BIT         EQU   0000000000001000B
WC_16BIT        EQU   0000000000010000B
WC_BALIGN       EQU   0000000000100000B
WC_POW2         EQU   0000000001000000B

WC_FETCHABLE    EQU   0000000010000000B ; InquireBridgeWindow
WC_CACHABLE     EQU   0000000100000000B ; InquireBridgeWindow

; ----- Memory window (page) capabilities only

WC_CALIGN       EQU   0000000010000000B
WC_PAVAIL       EQU   0000000100000000B
WC_PSHARED      EQU   0000001000000000B
WC_PENABLE      EQU   0000010000000000B
WC_WP           EQU   0000100000000000B

; ----- I/O window capabilities only

WC_INPACK       EQU   0000000010000000B
WC_EISA         EQU   0000000100000000B
WC_CENABLE      EQU   0000001000000000B

; ----- Generic window state

WS_IO           EQU   00000001B
WS_ENABLED      EQU   00000010B
WS_16BIT        EQU   00000100B   ; Memory and I/O only
```

```
; ----- Bridge window state

WS_PREFETCH    EQU   00001000B
WS_CACAHBLE    EQU   00011000B   ; Includes WS_PREFETCH

; ----- Memory window state

WS_PAGED       EQU   00001000B

; ----- I/O window state

WS_EISA        EQU   00001000B
WS_CENABLE     EQU   00010000B

; ----- Page state

PS_ATTRIBUTE   EQU   00000001B
PS_ENABLED     EQU   00000010B
PS_WP          EQU   00000100B

; ----- IRQ level bit-masks (low word of 32-bit mask)

IRQ_0    EQU   0000000000000001B
IRQ_1    EQU   0000000000000010B
IRQ_2    EQU   0000000000000100B
IRQ_3    EQU   0000000000001000B
IRQ_4    EQU   0000000000010000B
IRQ_5    EQU   0000000000100000B
IRQ_6    EQU   0000000001000000B
IRQ_7    EQU   0000000010000000B
IRQ_8    EQU   0000000100000000B
IRQ_9    EQU   0000001000000000B
IRQ_10   EQU   0000010000000000B
IRQ_11   EQU   0000100000000000B
IRQ_12   EQU   0001000000000000B
IRQ_13   EQU   0010000000000000B
IRQ_14   EQU   0100000000000000B
IRQ_15   EQU   1000000000000000B

; ----- IRQ level bit-masks (high word of 32-bit mask)

IRQ_NMI        EQU   0000000000000001B
IRQ_IO         EQU   0000000000000010B
IRQ_BUSERR     EQU   0000000000000100B

; ----- IRQ state bit-masks

IRQ_HIGH       EQU   01000000B
IRQ_ENABLED    EQU   10000000B
```

```
; ----- Socket bit-masks

SBM_WP          EQU   00000001B
SBM_LOCKED      EQU   00000010B
SBM_EJECT       EQU   00000100B
SBM_INSERT      EQU   00001000B
SBM_BVD1        EQU   00010000B
SBM_BVD2        EQU   00100000B
SBM_RDYBSY      EQU   01000000B
SBM_CD          EQU   10000000B

SBM_LOCK        EQU   00010000B
SBM_BATT        EQU   00100000B
SBM_BUSY        EQU   01000000B
SBM_XIP         EQU   10000000B

; ----- EDC definitions

EC_UNI          EQU   00000001B
EC_BI           EQU   00000010B
EC_REGISTER     EQU   00000100B
EC_MEMORY       EQU   00001000B
EC_PAUSABLE     EQU   00010000B

EC_WRITE        EQU   00000010B

ET_CHECK8       EQU   00000001B
ET_SDLC16       EQU   00000010B
ET_SDLC32       EQU   00000100B

; ----- Voltage Control values

VCTL_CISREAD    EQU   00010000B
VCTL_OVERRIDE   EQU   00100000B

VCTL_SENSE_MSK EQU   11000000b   ; Used to isolate voltage sense
VCTL_50V        EQU   00000000b
VCTL_33V        EQU   01000000b
VCTL_XXV        EQU   10000000b

; ----- Interface bit-masks

IF_TYPE_MASK    EQU   00000011B   ; Get/SetSocket
IF_CARDBUS      EQU   00000000B   ; GetSocket
IF_MEMORY       EQU   00000001B   ; Get/Inquire/SetSocket
IF_IO           EQU   00000010B   ; Get/Inquire/SetSocket
IF_CUSTOM       EQU   00000011B   ; Get/SetSocket

IF_CB           EQU   00000100B   ; InquireSocket
IF_DMA          EQU   00001000B   ; InquireSocket
IF_VSKEY        EQU   00010000B   ; InquireSocket
IF_33VCC        EQU   00100000B   ; InquireSocket
IF_XXVCC        EQU   01000000B   ; InquireSocket
```

```
DREQ_MASK       EQU   00001100B   ; Get/SetSocket
DREQ_NONE       EQU   00000000B   ; Get/SetSocket
DREQ_SPKR       EQU   00000100B   ; Get/SetSocket
DREQ_IOIS16     EQU   00001000B   ; Get/SetSocket
DREQ_INPACK     EQU   00001100B   ; Get/SetSocket

DMA_CHAN_MASK   EQU   11110000B   ; Get/SetSocket
DMA_CHAN0       EQU   00000000B   ; Get/SetSocket
DMA_CHAN1       EQU   00010000B   ; Get/SetSocket
DMA_CHAN2       EQU   00100000B   ; Get/SetSocket
DMA_CHAN3       EQU   00110000B   ; Get/SetSocket
DMA_CHAN4       EQU   01000000B   ; Get/SetSocket
DMA_CHAN5       EQU   01010000B   ; Get/SetSocket
DMA_CHAN6       EQU   01100000B   ; Get/SetSocket
DMA_CHAN7       EQU   01110000B   ; Get/SetSocket
DMA_CHAN8       EQU   10000000B   ; Get/SetSocket
DMA_CHAN9       EQU   10010000B   ; Get/SetSocket
DMA_CHAN10      EQU   10100000B   ; Get/SetSocket
DMA_CHAN11      EQU   10110000B   ; Get/SetSocket
DMA_CHAN12      EQU   11000000B   ; Get/SetSocket
DMA_CHAN13      EQU   11010000B   ; Get/SetSocket
DMA_CHAN14      EQU   11100000B   ; Get/SetSocket
DMA_CHAN15      EQU   11110000B   ; Get/SetSocket
```