Contents

LIST OF TABLES

1. GENERAL

2. SCOPE

3. CARD PHYSICAL

4. CARD INTERFACE

5. CARD METAFORMAT

6. EXECUTE IN PLACE (XIP)

A. GLOSSARY

INDEX

List of Tables

Contents

List of Tables

GENERAL

SCOPE

CARD PHYSICAL

CARD INTERFACE

CARD METAFORMAT

EXECUTE IN PLACE (XIP)

GLOSSARY

INDEX

Contents

LIST OF TABLES

1. GENERAL

2. SCOPE

3. CARD PHYSICAL

GENERAL

1.1 Introduction

The Personal Computer Memory Card International Association (PCMCIA) was formed with the goal of promoting interchangeability of Integrate Circuit Cards (IC Cards) among a variety of computer and other electronic products. Both data storage ("Memory") and peripheral expansion ("I/O") card types are defined by this standard. Many implementations will permit either type of 68-pin card to operate in the same slot. This standard governs a 68-pin interchange or I/O type card only. Cards compliant with this standard are referred to as *PC Cards*.

60-pin and 88-pin Dynamic RAM (DRAM) cards are standardized primarily by EIA/JEDEC and JEIDA, and are not covered by this standard. DRAM cards are used primarily as main memory expansion, and are not used for data interchange among systems.

PCMCIA serves both as a technical standards setting body and a trade association. To address these areas, the association operates both Technical and Marketing committees. The Technical Committee is concerned with the standard itself, and the Marketing Committee is primarily concerned with market development and promotional activities. It is the stated charter of PCMCIA to be market driven in its development of technical standards.

1.2 Relationship to Other Standard-Setting Bodies

Corresponding documents have been issued in the U.S., by PCMCIA, and in Japan, by the Japan Electronics Industry Development Association (JEIDA). The two organizations work closely together to ensure compatibility between the U.S. and Japanese releases of the standard.

The electrical and physical aspects of the initial release (1.0) of this standard are also recognized by the Electronics Industry Association (EIA) JEDEC JC-42 and JC-11 committees which standardize memory modules and packaging. JEDEC recognizes and endorses the software component of release 1.0, but does not publish those sections. This release, 2.0, is being taken to EIA/JEDEC, but its approval is still pending.

1.3 History

The basis of this standard is the 68-pin "2-piece connector" pin and socket type of Integrated Circuit Card, referred to herein as the *PC Card*. This form factor was originally defined by JEIDA in 1985. Prior to the publication in September 1990 of Release 1.0 of this standard, the existence and distribution of many proprietary and incompatible IC cards presented a major impediment to market acceptance.

1.4 Standardization Goals

The primary goal of this standard is to enable system and card manufacturers to build products operable by end-user's unfamiliar with the underlying technology. In order to satisfy this goal, the specific requirements (Electrical, Physical and Software) of numerous memory technologies, plug-in peripheral devices and the needs of various card applications had to be satisfied.

The standard was initially focused on IBM PC-compatible systems. A major goal of the standard, however, is to allow a variety of computer types and non-computer consumer products to freely interchange these cards. This standard is also applicable to embedded systems to the extent that the embedded system needs to be able to transfer its data back and forth to a host computer.

1.5 Differences Between Release 1.0 and 2.0

In release 1.0, all required hardware and software aspects required to build in *"Standard PC"* functionality were included. Release 1.0 fully defined issues relating to handling memory cards used as data storage devices in various environments.

In the electrical and software areas, Release 2.0 adds the software specification for *"XIP"*, the eXecute In Place mechanism which allows PCs to execute programs *directly from a Memory Card*. This mechanism frees up the system's RAM for data and "I/O," the hardware and software protocols which will allow *Credit-Card-Sized Peripherals*, such as network adapters and FAX modems, to be inserted in the *same card slots* as the memory cards. Other additions extend support for dual-voltage memory cards and add geometry and interleaving tuples, primarily used in support of Flash-type memory. Various editorial clarifications and corrections are also incorporated.

In the card's physical area, Release 2.0 adds a section dealing with card environmental requirements and test methods. Additionally, some connector parameters, including contact resistance, have been modified per discussions with other standards groups.

SECTION - 2

SCOPE

SCOPE

2.1 Elements of the Standard: Physical, Interface, Software

The goals of PC Card interchangeability are achieved by dividing the standard into three major sections. These are: *Card Physical, Card Interface* and *Card Data Format*. A brief overview of each of these sections is provided below.

2.2 Card Physical

This section spells out the dimensions and mechanical tolerances for memory cards and connectors. Specific pin lengths are defined to ensure that power is applied first and removed last during card insertion and removal. Reliability factors, such as connector mate/unmate cycles, environmental operating conditions and test methods are also specified. Two different card thicknesses are allowed: The more common Type I (3.3 mm thick) and the Type II (5.0 mm thick). In both types, connectors, guides and other factors are identical.

2.3 Card Interface

The card interface (electrical) section provides detailed pinout and signal definitions for both Memory- and I/O-type cards. Detailed functional and timing information is provided including the provision for reading 16-bit data on the low-order 8 data bits (useful in 8-bit host systems) and the interpretation of status information returned by the card.

Operations for both memory- and I/O-interface technologies are defined. The principal aspects of the card interface are byte addressability, random access to bytes of data, and the existence of a separate "register" attribute memory space selected by the REG signal. This allows a system to obtain highly detailed manufacturer and chip-type information from a card without the end user having to know or enter it. This also allows access to control registers in some configurable types of cards.

The interface has a 64-Mbyte addressing capability and numerous hardware provisions to support the various memory technologies, including ROM, OTPROM, UV-EPROM, Flash, SRAM and PSRAM. I/O-card support is provided by Interrupt, 16-bit cycle, IOread/IOwrite, INput ACK, Reset, Wait, Status Change, Enable and Power signals, some of which are dynamically redefined to these uses once an I/O card is recognized by the host.

2.4 Card Software

The software section deals with the organization of the data on the card. It accommodates both memory and configurable types of cards, and handles the I/O-card interface in a consistent manner. This information is conveyed to the host system via a *Metaformat* header also called the *Card Information Structure* (CIS). The Metaformat is the header at the beginning of the card which describes the low-level organization of the data on the card as well as the addressing and control requirements for I/O cards.

Various types of memory chips (ROM, SRAM, Flash, etc.) have different write and erase properties. In order to make most effective use of each type, different *data structures* or organizing schemes are needed for each type. Certain card applications, such as XIP, or proprietary, dedicated systems also impose data organization constraints. The Metaformat allows system designers to choose the appropriate format and still maintain interchangeability among systems. The Metaformat also conveys card architectural information such as data blocking or interleaving requirements imposed by a card's hardware.

SECTION - 3

CARD PHYSICAL

CARD PHYSICAL

This section of the specification defines the PC Card's physical outline dimensions, connector system and qualification test parameters. The test specified in subsections 3.4 Connector Reliability, 3.5 Connector Durability and 3.6 PC Card Environmental are the minimum parameters which must be met.

Each manufacture will qualify their products using this specification confirmed by recognized standard qualification test procedures.

3.1 Card Dimensions

There are two types of PC Cards in this specification. They are Type I (Figure 3-1) and Type II (Figure 3-2). The two PC Card types differ only in thickness (Figures 3-3 and 3-4). Type II PC Card thickness is greater than Type I in the substrate area (Figures 3-2 and 3-4). Type I PC Card is preferred and Type II PC Card is optional.

The PC Card dimensions for the Type I and Type II are shown in Table 3-1.

Table 3-1: PC Card Dimensions

	LENGTH (±.008)	WIDTH (±.004)	INTERCONNECT AREA (±.002)[1]	SUBSTRATE AREA (±.004)[1]
TYPE I	3.370 (85.6)	2.126 (54.0)	.065 (1.65)	.065 (1.65)
TYPE II	3.370 (85.6)	2.126 (54.0)	.065 (1.65)	.098 MAX (2.5)

1. Interconnect area and substrate area thickness are specified from the PC Card center line to either the top or bottom surface.
2. Millimeters are shown in parentheses ( ).

Connector location and pin numbers for Type I and Type II PC Cards are shown in Figures 3-1 and 3-2. PC Card polarization technique and dimensions are also shown in Figures 3-1 and 3-2. A mismated PC Card and connector shall withstand a minimum static load of 13 pounds (6 kg) without damage to the PC Card or connector.

PC Cards must be opaque (non see-through).

3.1.1 Write Protect Switch (WPS)

The WPS, if installed, shall be located to the right of the PC Card centerline when viewed from the end opposite the connector (Figures 3-1, 3-2, 3-3 and 3-4).

The write-protected position of the WPS shall be the far-right position, and shall be indicated by an arrow and the words "Write Protect" or "Protect". The arrow and indication may be on the end of the PC Card, as shown in Figures 3-1, 3-2, 3-3 and 3-4, or on the bottom cover, as shown in Figure 5, or on both the end and bottom cover.

If a WPS is used, it is recommended that it pass all requirements, as applicable, in subsection 3.6 PC Card Environment. It is also recommended the WPS function as specified for a minimum of 100 (Write Protect/Write Enable) cycles.

3.1.2 Battery Location

The battery, if installed, should be located to the left of the PC Card centerline when viewed from the end opposite the connector (Figures 3-1, 3-2, 3-3 and 3-4).

The battery holder, if installed, should be designed so that the positive (+) side of the battery faces the top surface.

3.1.3 Label

Use of the label is optional. If used, the label shall be located on the bottom cover (Surface B - see Figures 3-1 and 3-2).

The thickness of the label, if used, (Figure 3-5) shall not cause the PC Card to exceed the thickness specified in Figure 3-1, 3-2 and Table 3-1.

The label, if used, must withstand all environmental test specified in subsection 3.6 PC Card Environmental.

The JEIDA and PCMCIA logos may be displayed by the manufacturer if authorized by the respective organizations. Logo location is shown in Figure 3-5.

3.2 Connector

The specified PC Card interconnect system shall be a 68-position, 2-piece pin-and-socket. The socket contacts shall be the PC Card connector.

3.2.1 Card Connector

The socket contacts are located on the PC Card as shown in Figures 3-1, 3-2, 3-3 and 3-4. The PC Card connector socket shall be configured as shown in Figure 3-6.

The PC Card connector socket contacts shall make contact with the connector pin for a minimum length of 0.050 (1.27) as shown in Figure 3-7.

The PC Card connector socket layout shall match the host pin-connector layout as shown in Figure 3-8.

3.2.2 Host Connector

The host pin connector shall be a 68-pin connector with opening, polarization and pin location as shown in Figure 3-8. The host connector-pin configuration is shown in Figure 3-9, and the host-pin lengths are shown in Figure 3-9 and Table 3-2.

Table 3-2: Host Connector Pin Configuration

Pin Type	Pin Length (L) ±.004	Pin Number
Detect	.138 (3.5)	36, 67
General	.167 (4.25)	All Other Pins
Power	.197 (5.0)	1, 17, 34, 35, 51, 68

The outermost plating of socket and pin contact area shall be gold, or other plated materials compatible with gold, and shall meet the requirements specified in subsections 3.4 and 3.5.

The recommended host connector PCB footprints for: the right angle connector (Figure 3-10), the straight connector (Figure 3-11), two row surface mount (Figure 3-12), one row surface mount (Figure 3-13) and double [136 position] surface mount (Figure 3-14) are shown without mounting or hardware hole locations.

The interconnect system shall pass all requirements of subsection 3.4 (Connector Reliability) and subsection 3.5 (Connector Durability).

If a connector ejector mechanism is used, it is recommended the ejector mechanism pass all requirements for reliability and durability, as applicable, in subsections 3.4 and 3.5.

3.3 PC Card Guidance

The PC Card shall be guided by the host connector for a minimum distance of 0.394" (10.0) before the socket connector bottoms on the host (pin) connector (Figure 3-15).

To ensure alignment of the PC Card to connectors, the PC Card should be guided for a minimum distance of 1.570" (40.0) before engagement.

3.4 Connector Reliability

The interconnect system as specified in subsection 3.2 shall meet or exceed all reliability test requirements of this subsection. Unless otherwise specified, all test and measurements shall be made at:

Temperature	15°C to 35°C
Air pressure	650 to 800 mm mercury (860 to 1060 mbar)
Relative humidity	25% to 85%

If conditions must be closely controlled in order to obtain reproducible results, the parameters shall be:

Temperature	23°C +/- 1°C
Air pressure	650 to 800 mm mercury (860 to 1060 mbar)
Relative humidity	50% +/- 2%

3.4.1 Mechanical Performance

The interconnect system mechanical performance is specified as follows:

3.4.1.1 Office Environment

STANDARD	TESTING
Guaranteed number of insertions/ejections =10,000 min.	Paragraph 3.5.1

3.4.1.2 Harsh Environment

STANDARD	TESTING
Guaranteed number of insertions/ejections = 5,000 min.	Paragraph 3.5.2

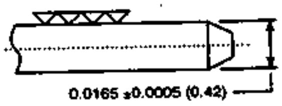
3.4.1.3 Total Insertion Force

STANDARD	TESTING
8.8lbs (4kg) max.	Insert and extract at speed of 1" (25mm)/min.

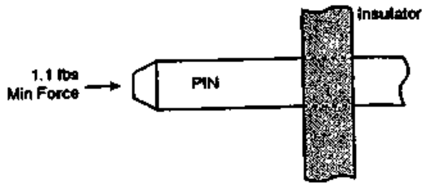
3.4.1.4 Total Pulling Force

STANDARD	TESTING
1.5lbs (.68kg) min.	Insert and extract at speed of 1" (25mm)/min.

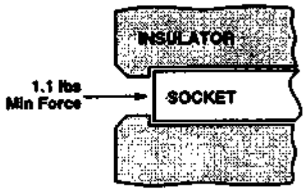
3.4.1.5 Single Pin Pulling Force

STANDARD	TESTING
0.022lbs (10g) minimum initial value only.  Gauge: Material - Tool making steel Hardness - HRC = 50 to 55	Pull the gauge pin shown to left at speed of 1" (25mm)/min. Gauge pin's surface must be wiped clean of dirt and lubrication oil.

3.4.1.6 Single Pin Holding Force

STANDARD	TESTING
Pin shall not push out of the insulator when 2.2lbs (1kg) minimum force is applied to the pin.	Push pin on the axis with 2.2 lbs (1kg) minimum force while holding insulator rigid. 

3.4.1.7 Single Socket Holding Force

STANDARD	TESTING
Socket shall not push out of the insulator when 1.1 lbs (0.5 kg) minimum force is applied to the socket.	Push socket on the axis with 1.1 lbs (0.5kg) minimum force while holding insulator rigid. 

3.4.1.8 Vibration and High Frequency

STANDARD	TESTING
a. No mechanical defects should occur on the parts. b. Must not cause current interruption of 100ns or more.	MIL-STD-202F METHOD 204D Test condition B (15G peak), 10Hz to 2000Hz; (Figure 3-16)

3.4.1.9 Shock

STANDARD	TESTING
a. No mechanical defects should occur on the parts. b. Must not cause current interruption of 100ns or more	MIL-STD-202F METHOD 213B, Acceleration 50G, Standard holding time 6ms, Semi-sine wave; (Figure 3-16)

3.4.2 Electrical Performance

The interconnect system electrical performance is specified as follows.

3.4.2.1 Contact Resistance (low level)

STANDARD	TESTING
a. Initially... 40 mΩ maximum b. After test... 20 mΩ maximum change.	MIL-STD-1344A METHOD 3002.1 Open voltage 20 mV Test current 1mA a. Measure and record the initial resistance (Ri) of the connector system (from attachment of the pin connector to the PCB, to the attachment of the socket connector to the PCB) Ri ≤ 40 mΩ b. Measure and record resistance of the connector system after test (R_f) of the connector system. Resistance value after test: R_f - Ri ≤ 20 mΩ

3.4.2.2 Withstandable Voltage

STANDARD	TESTING
a. No shorting or other damages when 500 Vrms AC is applied for 1 minute b. Current leakage 1 mA max.	MIL-STD-202F METHOD 301

3.4.2.3 Insulation Resistance

STANDARD	TESTING
a. Initially... 1,000 MΩ min. b. After test... 100MΩ min	MIL-STD-202F METHOD 302 Measure within 1 minute after applying 500V DC

3.4.2.4 Current Capacity

STANDARD	TESTING
0.5 A per pin	

3.4.2.5 Insulation Material

STANDARD	TESTING
UL 94 V-0 equivalent	

3.4.3 Environmental Performance

3.4.3.1 Operating Environment

STANDARD	TESTING
Operating Temperature: -20°C to +60°C Relative humidity: 95% Max. (non-condensing)	

3.4.3.2 Storage Environment

STANDARD	TESTING
Storage Temperature: -40°C to +70°C Relative humidity: 95% Max. (non-condensing)	

3.4.4 Environmental Resistance

3.4.4.1 Moisture Resistance

STANDARD	TESTING
Contact resistance: 3.4.2.1.b. Insulation resistance: 3.4.2.3.b.	MILSTD-202F METHOD 106E (excluding vibration); 10 cycles (1 cycle = 24 hours) with connector engaged.

3.4.4.2 Thermal Shock

STANDARD	TESTING
No physical damage should occur during testing. Contact resistance: 3.4.2.1.b. Insulation resistance: 3.4.2.3.b.	MIL-STD-202F METHOD 107G-Test condition A, -55°C to +70°C 5 cycles (1 cycle = 1 hour) with connector engaged

3.4.4.3 Durability (High Temperature)

STANDARD	TESTING
Contact resistance: 3.4.2.1.b.	MIL-STD-202F METHOD 108A Test condition B, 85°C, 250 hours min. with connector engaged

3.4.4.4 Cold Resistance

STANDARD	TESTING
Contact resistance: 3.4.2.1.b.	JIS C 5021, -55°C, 96 hours min. with connector engaged

3.4.4.5 Humidity (Normal Condition)

STANDARD	TESTING
Contact resistance: 3.4.2.1.b. Insulation resistance: 3.4.2.3.b.	MIL-STD-202F METHOD 103B Test condition B, 40°C, 90 to 95% RH with connector engaged

3.4.4.6 Hydrogen Sulfide

STANDARD	TESTING
Contact resistance: 3.4.2.1.b.	JEIDA 38 3ppm hydrogen sulfide 40°C, approx 80% RH 96 hours min., with connector engaged

3.4.4.7 Salt Water Spray

STANDARD	TESTING
No harmful corrosion (or degradation of contact performance) should occur on the pin and socket contacts.	MIL-STD-202F METHOD 101D Test condition B, Concentration 5% 35°C, 48 hours, with connector disengaged

3.5 Connector Durability

The interconnect system as specified in subsection 3.2 shall meet or exceed all durability requirements of this subsection.

Test conditions for the mate/unmate cycles are:

Cycle Rate	400-600 cycles per hour
Temperature	15°C to 35°C
Relative Humidity	25%-85%
Barometric Pressure	650 to 800 mm mercury (860 to 1060 mbar)

3.5.1 Office Environment

The office environment is defined in EIA-364A as class 1.1 - year round air conditioning (non-filtered) with humidity control.

Test Sequence

Contact resistance per 3.4.2.1.a.
Mate and unmate the connector for a total of 10,000 cycles
Contact resistance per 3.4.2.1.b.

3.5.2 Harsh Environment

The harsh environment is defined in EIA-364A as class 1.3 - no air conditioning, no humidity control with normal heating and ventilation.

Contact resistance per 3.4.2.1.a.	
Mate and unmate the connector 1,000 cycles	TOTAL CYCLES = 1,000
Humidity per 3.4.4.5.	
Mate and unmate the connector 1,000 cycles	TOTAL CYCLES = 2,000
Humidity per 3.4.4.5.	
Mate and unmate the connector 3,000 cycles	TOTAL CYCLES = 5,000
Humidity per 3.4.4.5.	
Hydrogen sulfide per 3.4.4.6.	
Contact resistance per 3.4.2.1.b.	

3.6 PC Card Environmental

The PC Card as specified in Section-3 CARD PHYSICAL shall meet or exceed all environmental tests of subsection 3.6.2. The battery, if part of the PC Card, shall be installed for all tests. Unless otherwise specified, all test and measurements shall be made at:

Temperature	15°C to 35°C
Air Pressure	650 to 800 mm mercury (860 to 1060 mbar)
Relative Humidity	25% to 85%

If conditions must be closely controlled in order to obtain reproducible results, the parameters shall be:

Temperature	23°C + /- 1°C
Air Pressure	650 to 800 mm mercury (860 to 1060 mbar)
Relative Humidity	50% ±2%

3.6.1 Environmental Performance

The PC Card storage and operating environment are specified in this subsection. The storage and operating environment test parameters are specified in subsection 3.6.2 Environmental Resistance.

3.6.1.1 Operating Environment

STANDARD	TESTING
Ambient temperature 0°C to 55°C	
Relative humidity 95% max. (non condensing)	
SRAM data retention per 3.6.2.18.	

3.6.1.2 Storage Environment

STANDARD	TESTING
Storage temperature -20°C to 65°C	
Relative humidity 95% max. (non condensing)	
SRAM data retention per 3.6.2.18.	

3.6.2 Environmental Resistance

The PC Card shall be tested per the Environmental Resistance specifications listed below. The manufacturer shall ensure adequate testing in order to ensure the PC Card conforms to this specification.

3.6.2.1 High Storage Temperature

STANDARD	TESTING
PC Card to function as specified after test and all non-volatile memory to retain the data stored prior to test. SRAM data retention per 3.6.2.18.	MIL-STD-202F Method 108A Test Condition 65°C and 90% minimum humidity for 96 hours min, Vcc=0

3.6.2.2 Low Storage Temperature

STANDARD	TESTING
PC Card to function as specified after test and all non- volatile memory to retain the data stored prior to test. SRAM data retention per 3.6.2.18.	Test Condition -20°C for 96 hours minimum,Vcc=0

3.6.2.3 High Operating Temperature

STANDARD	TESTING
PC Card to function as specified during test.	MIL-STD-202F Method 108A Test Condition 55°C for 96 hours minimum Vcc = manufacturer specified.

3.6.2.4 Low Operating Temperature

STANDARD	TESTING
PC Card to function as specified during test.	MIL-STD-202F Method 108A Test Condition 0°C for 96 hours minimum Vcc = manufacturer specified.

3.6.2.5 Thermal Shock

<table>
<tr><th>STANDARD</th><th>TESTING</th></tr>
<tr><td>PC Card to function as specified after test and all non- volatile memory to retain the data stored prior to test

SRAM data retention per 3.6.2.18</td><td>MIL-STD-202F
Method 107G
<table>
<tr><th>TEST</th><th>TEMP (C)</th><th>TIME (Min)</th></tr>
<tr><td>1</td><td>-20°C</td><td>30</td></tr>
<tr><td>2</td><td>25°C</td><td>&lt;5</td></tr>
<tr><td>3</td><td>65°C</td><td>30</td></tr>
<tr><td>4</td><td>25°C</td><td>&lt;5</td></tr>
</table>
Repeat for 100 cycles Vcc=0, Card connector engaged.</td></tr>
</table>

3.6.2.6 Moisture Resistance

<table>
<tr><th>STANDARD</th><th>TESTING</th></tr>
<tr><td>PC Card to function as specified during test.</td><td>MIL-STD-202F
Method 106E
<table>
<tr><th>STEP</th><th>TEMP (C)</th><th>RH(%)</th><th>TIME (Hrs)</th></tr>
<tr><td>1</td><td>25 to 55</td><td>90-98</td><td>2.5</td></tr>
<tr><td>2</td><td>55</td><td>90-98</td><td>3.0</td></tr>
<tr><td>3</td><td>55 to 25</td><td>80-98</td><td>2.5</td></tr>
<tr><td>4</td><td>25 to 0</td><td>N/A</td><td>2.5</td></tr>
<tr><td>5</td><td>0</td><td>N/A</td><td>3.0</td></tr>
<tr><td>6</td><td>0 to 25</td><td>N/A</td><td>2.5</td></tr>
</table>
Repeat test for 10 cycles (excluding vibration)
Vcc = manufacturer specified, Card connector engaged.</td></tr>
</table>

3.6.2.7 Electrostatic Discharge

STANDARD	TESTING
PC Card to function as specified after test 1 and 2 and all non-volatile memory to retain the data stored prior to test	ISO 7816-1 TEST 1. Discharge four (4) times on the top cover. Repeat test on bottom cover. Total discharge cycles = 8 (Figure 3-17) TEST 2. Discharge total twelve (12) times each side as indicated in Figure 3-18. Turn PC Card over and repeat test.Total discharge cycles = 24.

3.6.2.8 X-ray Exposure

STANDARD	TESTING
PC Card to function as specified after test and all non- volatile memory to retain the data stored prior to test	ISO 7816-1 140 kv @ 5ma Intensity 10K roentgen Exposure time 5 minutes.

3.6.2.9 Ultraviolet Light Exposure

STANDARD	TESTING
PC Card to function as specified after test and all non- volatile memory to retain the data stored prior to test	ISO 7816-1 Wavelength 254 nm Intensity 15000 µW/cm Exposure time 20 minutes

3.6.2.10 Electromagnetic Field Interference

STANDARD	TESTING
PC Card to function as specified after test and all non- volatile memory to retain the data stored prior to test	ISO 7816-1 Place PC Card in uniform magnetic field of 1,000 Oersted Exposure time 10 seconds

3.6.2.11 Card Inverse Insertion

STANDARD	TESTING
No visible damage to PC Card or connector after test. No electrical contact to PC Card during test.	Insert PC Card with top side down. Exert a force of 13 lbs for 1 minute. Repeat 5 times

3.6.2.12 Vibration and High Frequency

STANDARD	TESTING
No visible damage to PC Card after test. PC Card to function as specified after test and all non-volatile memory to retain the data stored prior test	MIL-STD-202F Method 204D Test Condition B, 15G peak, 10 to 2,000 Hz, Vcc=0, With battery installed (Figure 3-19)

3.6.2.13 Shock

STANDARD	TESTING
No visible damage to PC Card after test. PC Card to function as specified after test and all non-volatile memory to retain the data stored prior to test	MIL-STD-202F Method 213B Test Condition A (Figure 3-19)

3.6.2.14 Bend Test

STANDARD	TESTING
No visible damage to PC Card after test. PC Card to function as specified after test and all non-volatile memory to retain the data stored prior to test	Test 1: Clamp the connector end of the PC Card with surface A facing upward. Apply 4.4 lbs (2 kg) to the unclamped end using the force bar as shown in Figure 3-20. Time ≥ 1 minute. Test 2: Clamp the non-connector end of the PC Card with surface A facing upward. Apply 4.4 lbs (2 kg) to the unclamped end using the force bar as shown in Figure 3-20. Time ≥ 1 minute. Test 3 and 4: Repeat test 1 and 2 with surface B facing upward. Total test must include all four procedures.

3.6.2.15 Drop Test

STANDARD	TESTING
No visible damage to PC Card after test. PC Card to function as specified after test and all non-volatile memory to retain the data stored prior to test	Drop PC Card from 30" (75 cm) onto a non-cushioning, vinyl-tile surface. Repeat 6 times.

3.6.2.16 Torque Test

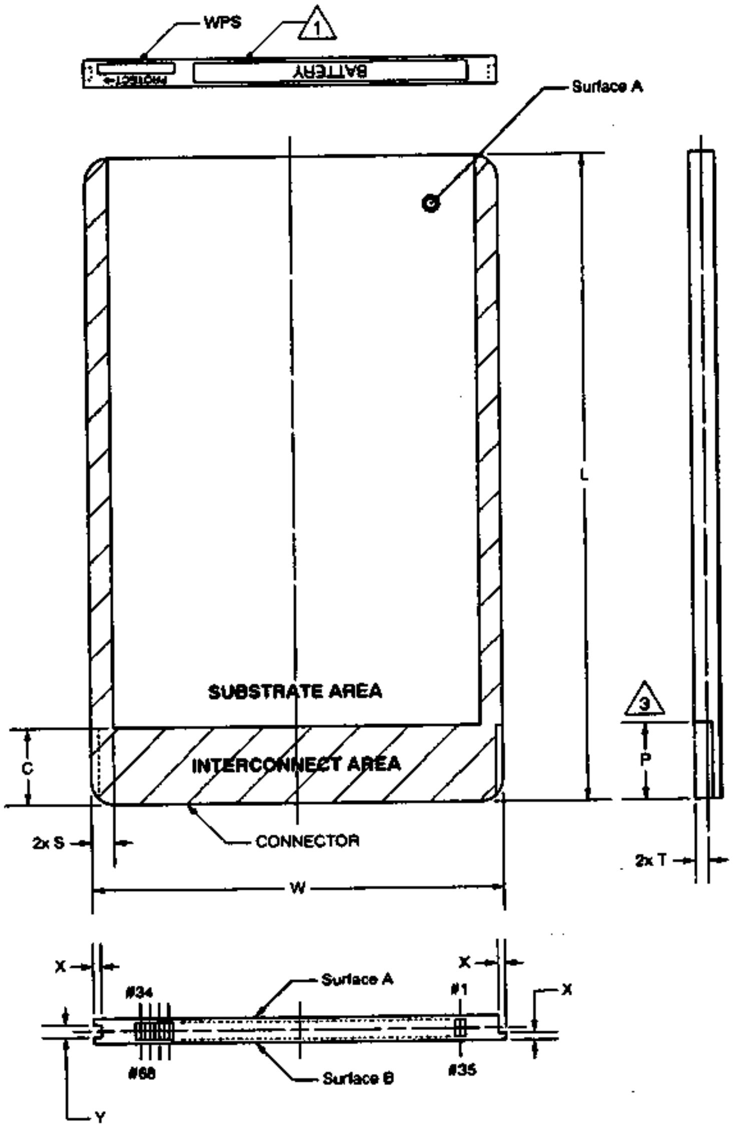
STANDARD	TESTING
No visible damage to PC Card after test. PC Card to function as specified after test and all non-volatile memory to retain the data stored prior to test	ISO 7816-1 Apply clock-wise torque to the unsupported end of the PC Card (torque = 11 in-lbs max. and/or angle = 10° max., whichever occurs first) Time = 5 minutes. Repeat test applying counter clock-wise torque. Repeat test five (5) times in each direction (Figure 3-21).

3.6.2.17 PC Card Warpage

STANDARD	TESTING
Type I PC Card shall exhibit warpage less than or equal to: Interconnect Area 1 ≤ .006" (.15) Interconnect Area 2 ≤ .014" (0.35) Substrate Area 3 ≤ .020" (.5) Substrate Area thickness shall not exceed .150" (3.8). Type II PC Card shall exhibit warpage less than or equal to: Interconnect Area 1 ≤ .006" (.15) Interconnect Area 2 ≤ .014" (0.35) Substrate Area 3 ≤ .020" (.5) Substrate Area thickness shall not exceed .211" (5.35).	Measure the PC Card (Type I or II) interconnect and substrate thicknesses. Then place the PC Card (Type I or II) on a flat plate and measure the maximum warpage (Figures 3-22 and 3-23).

3.6.2.18 SRAM Data Retention

STANDARD	TESTING
SRAM PC Card to retain all data after each test (1 and 2).	Test 1: Test condition 55°C for 24 hours min. Vcc=0 Test 2: Test condition 0°C for 24 hours min. Vcc=0



C MIN	L ± 0.008	P MIN △3	S MIN	T △4	W ± 0.004	X ± 0.002	Y ± 0.002
.394 (10.0)	3.370 (85.60)	.394 (10.0)	.118 (3.0)	.065 (1.65)	2.126 (54.0)	.039 (1.00)	.063 (1.60)

△1 RECOMMENDED BATTERY LOCATION. THE BATTERY HOLDER SHOULD BE DESIGNED SO THAT THE POSITIVE SIDE OF THE BATTERY IS UP

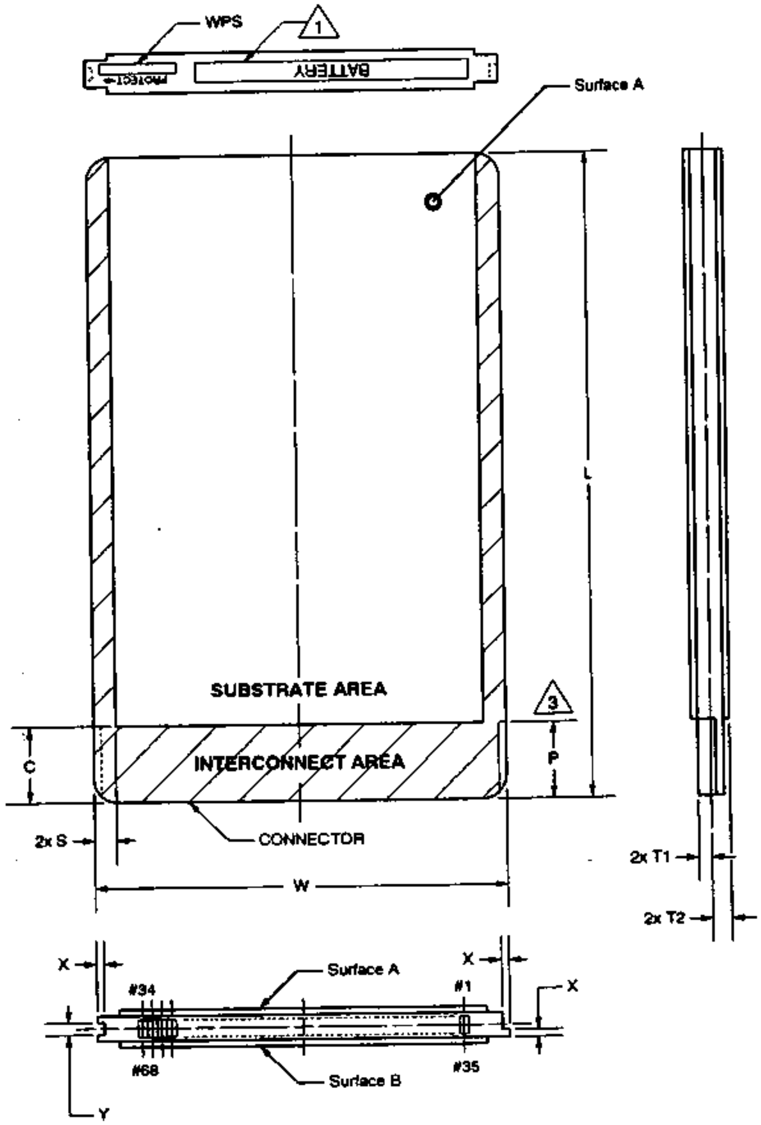
2 THE PC CARD SHALL BE OPAQUE (NON SEE THRU)

△3 POLARIZATION KEY LENGTH.

△4 INTERCONNECT AREA TOLERANCE = ±.002
SUBSTRATE AREA TOLERANCE = ±.004

5 MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-1. TYPE I PC Card Package Dimensions



C MIN	L ± 0.008	P MIN ⚠3	S MIN	T1 ±0.002	T2 MAX	W ± 0.004	X ± 0.002	Y ± 0.002
.394 (10.0)	3.370 (85.60)	.394 (10.0)	.118 (3.0)	.065 (1.65)	.098 (2.50)	2.126 (54.0)	.039 (1.00)	.063 (1.60)

⚠1 RECOMMENDED BATTERY LOCATION. THE BATTERY HOLDER SHOULD BE DESIGNED SO THAT THE POSITIVE SIDE OF THE BATTERY IS UP

2. THE PC CARD SHALL BE OPAQUE (NON SEE THRU)

⚠3 POLARIZATION KEY LENGTH.

4 MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-2. TYPE II PC Card Package Dimensions

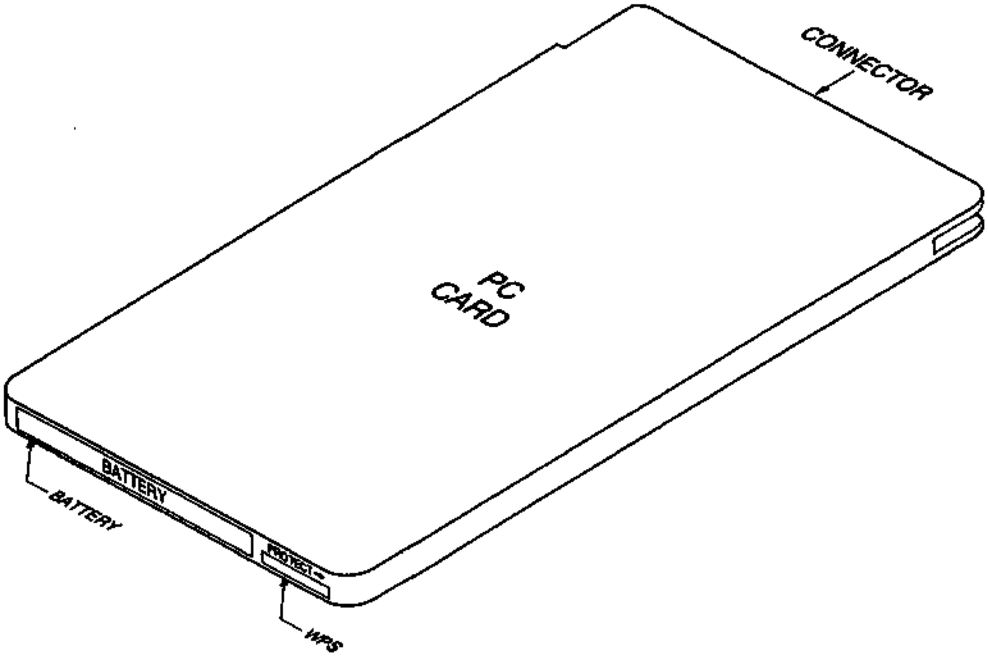


Figure 3-3. TYPE I PC Card

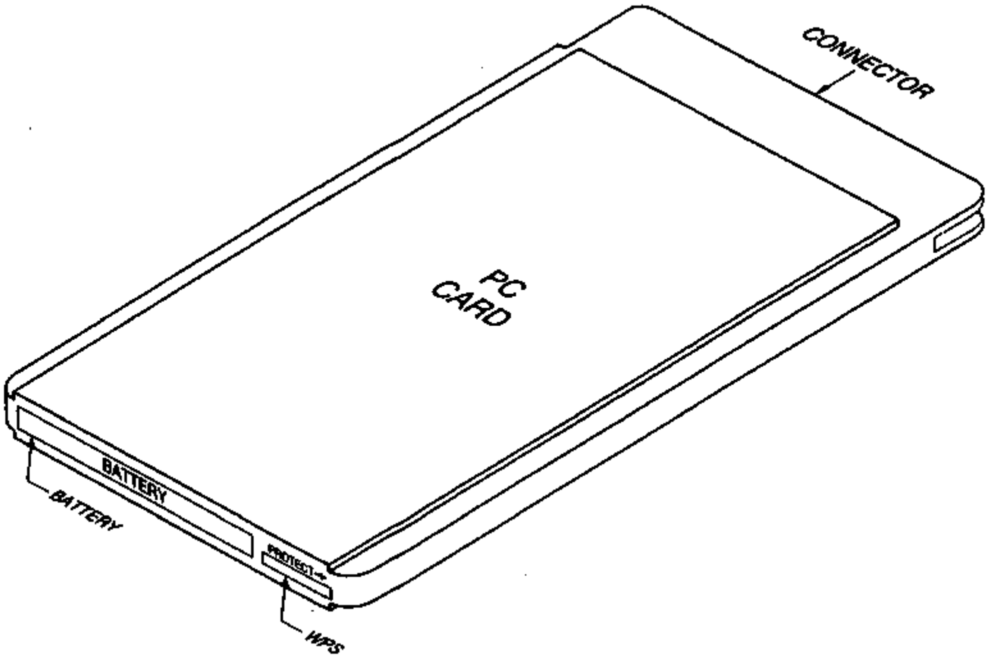
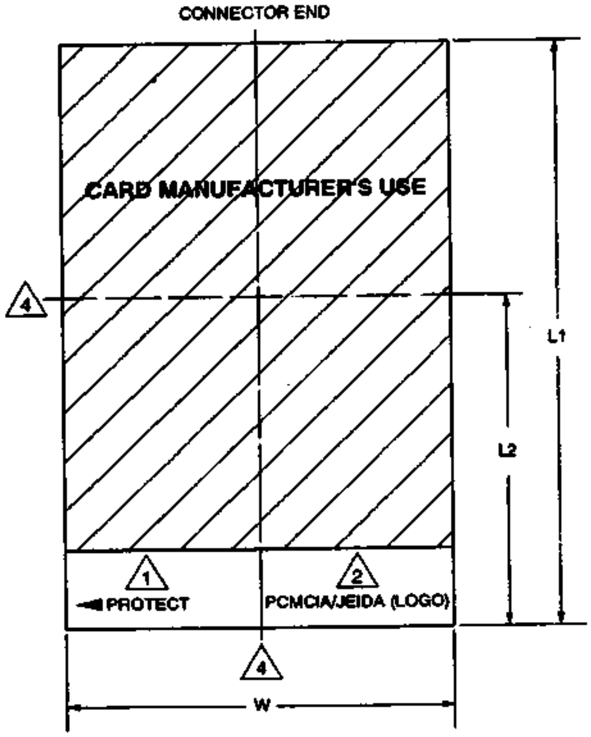


Figure 3-4. TYPE II PC Card



L1 Max	L2 Max	W Max
2.968 (75.39)	1.681 (42.70)	1.886 (47.90)

△1 IF WRITE PROTECT SWITCH INSTALLED

△2 OTHER REGULATORY LOGOS

3. LABELS MUST WITHSTAND ALL PC CARD ENVIRONMENTAL SPECIFICATIONS IN SUBSECTION 3.6

△4 PC CARD CENTERLINES

5 LABEL, IF USED, SHALL BE ATTACHED TO THE BOTTOM COVER (SURFACE 6 - FIGURES 3-1 AND 3-2).

6 MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-5. PC Card Bottom Cover Label

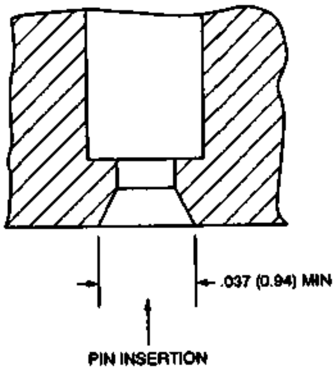
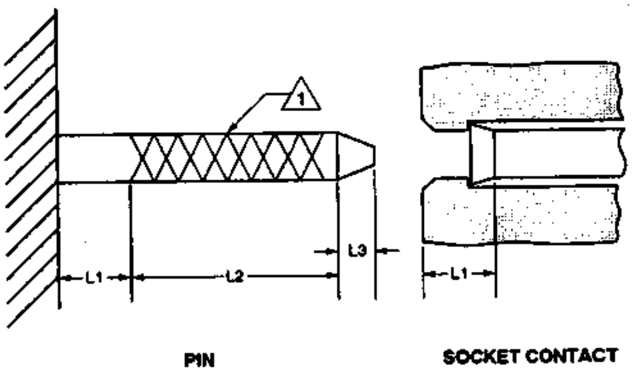


Figure 3-6. Card Connector Socket

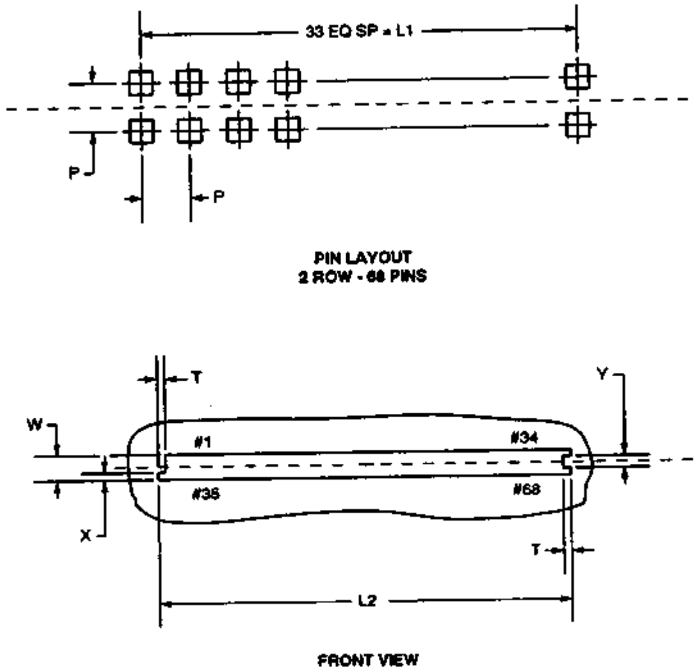


△1 PIN/SOCKET CONTACT AREA

2. MILLIMETERS ARE IN PARENTHESIS ( )

L1 MAX	L2	L3 REF
.020 (0.5)	PIN TYPE - SEE TABLE 3-2 DETECT .059 (1.5) ±.039 GENERAL .084 (2.1) ±.064 POWER .098 (2.5) ±.078	.024 (0.6)

Figure 3-7. Pin/Socket Contact length with Wipe

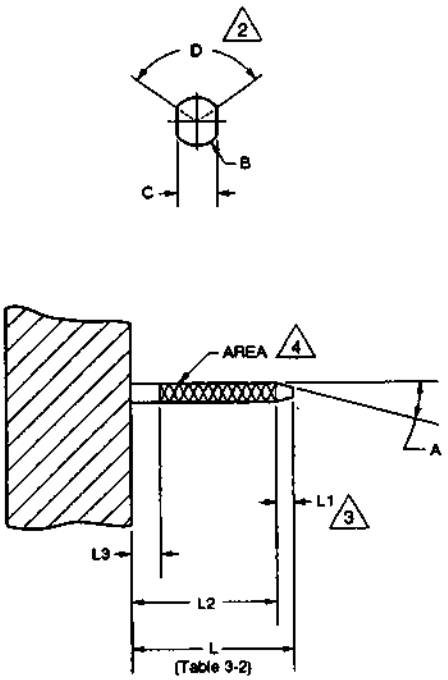


PIN LAYOUT
2 ROW - 68 PINS

FRONT VIEW

L1 ± .006	L2 ± .003	P .004	T ± .002	W ± .003	X ± .002	Y ± .002
1.65 (41.9)	2.135 (54.23)	.050 (1.27)	.035 (0.90)	.139 (3.53)	.047 (1.20)	.055 (1.40)

Figure 3-8. Pin Connector



A +5°/-0°	B ± .001 ⌀	C MAX	D MIN	L1 ± .004	L2 MIN	L3 MAX
10°	.017 (0.44)	.018 (0.46)	110°	.020 (0.50)	.114 (2.9)	.020 (0.50)

1 LENGTH "L" GIVEN IN TABLE 3-2

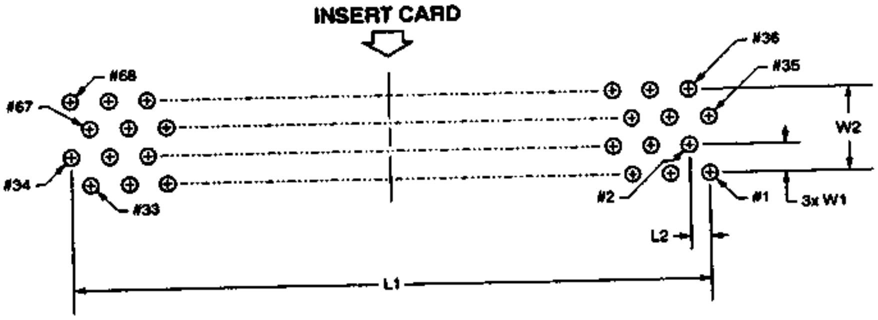
2 MINIMUM CONTACT AREA (2 PLACES)

3 PIN TAPER LENGTH

4 SOCKET CONTACT ENGAGEMENT AREA

5 MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-9. PC Card Contact Pins



L1 ± .003	L2 ± .002	W1 ± .002	W2 REF
1.650 (41.91)	.050 (1.27)	.075 (1.905)	.225 (5.715)

1. Millimeters are in parenthesis( ).

Figure 3-10. Recommended Right Angle Connector PCB Footprint

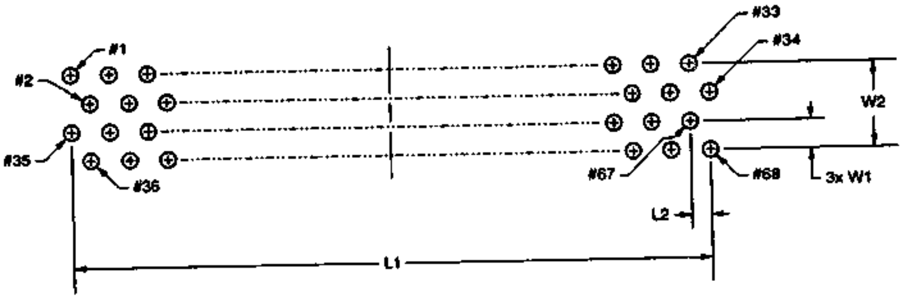
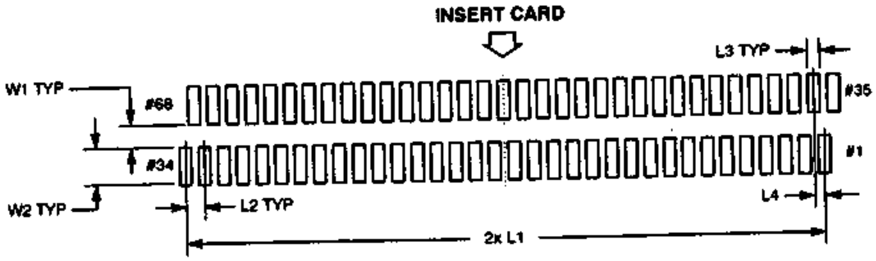


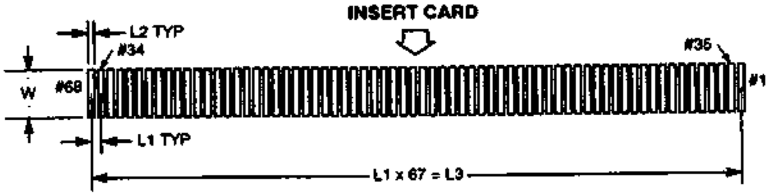
Figure 3-11. Recommended Straight Connector PCB Footprint



L1 ±. .004	L2 ± .002	L3 ± .002	L4 Ref	W1 ± .004	W2 ± .004
1.650 (41.91)	.050 (1.27)	.031 (.079)	.025 (0.635)	.059 (1.50)	.098 (2.50)

1. Millimeters are in parenthesis ( ).

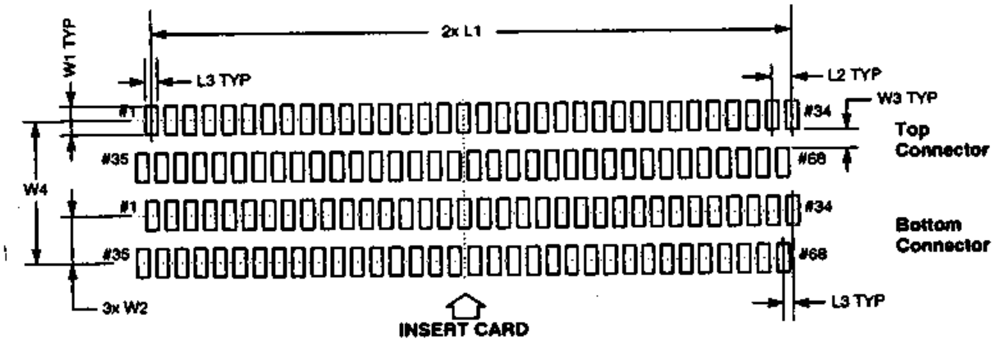
Figure 3-12. Recommended Surface Mount Connector PCB Footprint



L1 ± .002	L2 ± .002	L3 ± .002	W ± .002
.025 (0.635)	.016 (0.40)	1.675 (42.54)	.123 (3.10)

1. Millimeters are in parenthesis ( ).

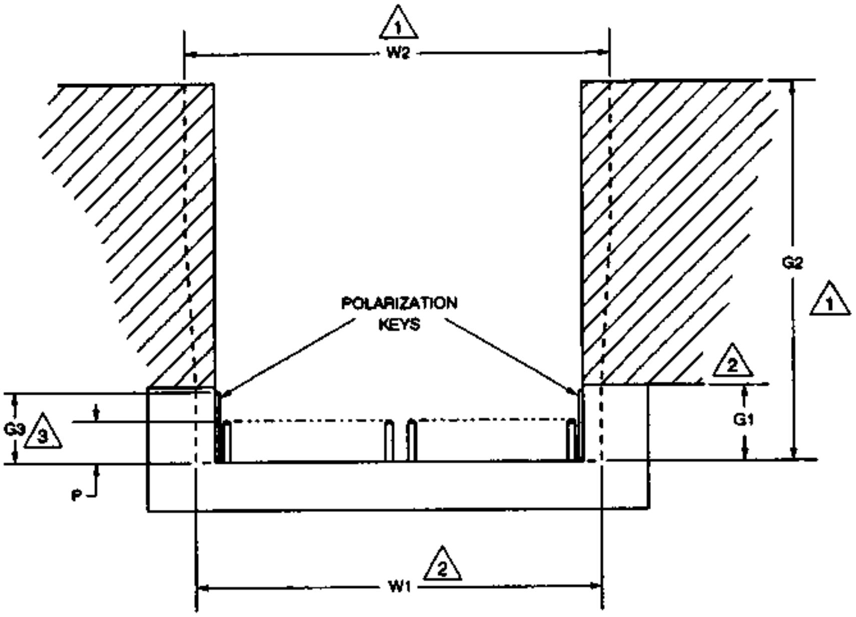
Figure 3-13. Recommended Surface Mount Connector PCB Footprint



L1 ± .004	L2 ± .002	L3 ± .002	W1 ± .004	W2 ± .002	W3 ± .002	W4 Ref
1.650 (41.91)	.050 (1.27)	.025 (0.635)	.075 (1.90)	.126 (3.20)	.051 (1.30)	.378 (9.60)

1. Millimeters are in parenthesis ( ).

Figure 3-14. Recommended Double (136 Position) Surface Mount P.C.B. Footprint



G1 MIN	G2 MIN	G3 ± .012	P MAX	W1 ± .003	W2 ± .005
.394 (10.0)	1.570 (40.0)	.382 (9.7)	.201 (5.1)	2.135 (54.23)	2.239 (55.60)

1 THE PC CARD SHOULD BE GUIDED FOR A MINIMUM DISTANCE OF 1.570 (40.0)

2 THE CONNECTOR MUST GUIDE THE PC CARD FOR A MINIMUM DISTANCE OF .197 (5.0) BEFORE ENGAGEMENT

3 THE CONNECTOR POLARIZATION KEYS ARE DEFINED IN FIGURE 3-8.

4 MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-15. PC Card Guide Guidance

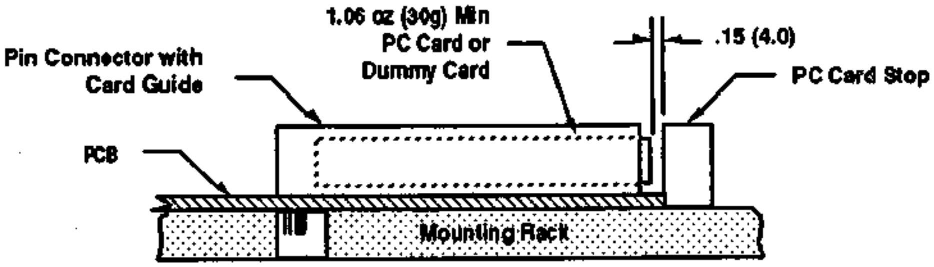
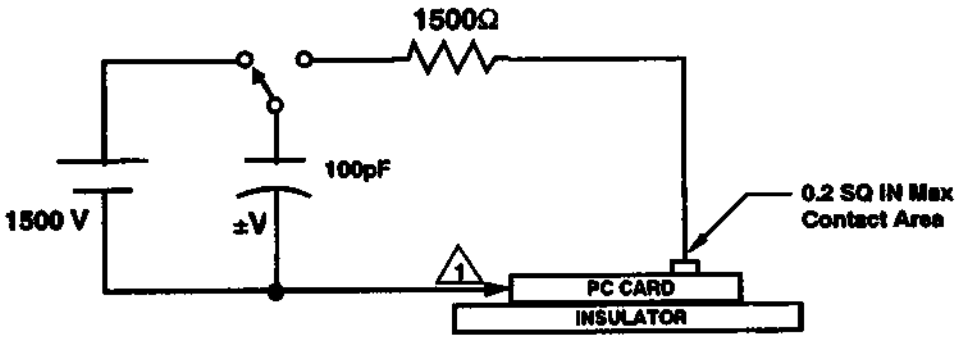
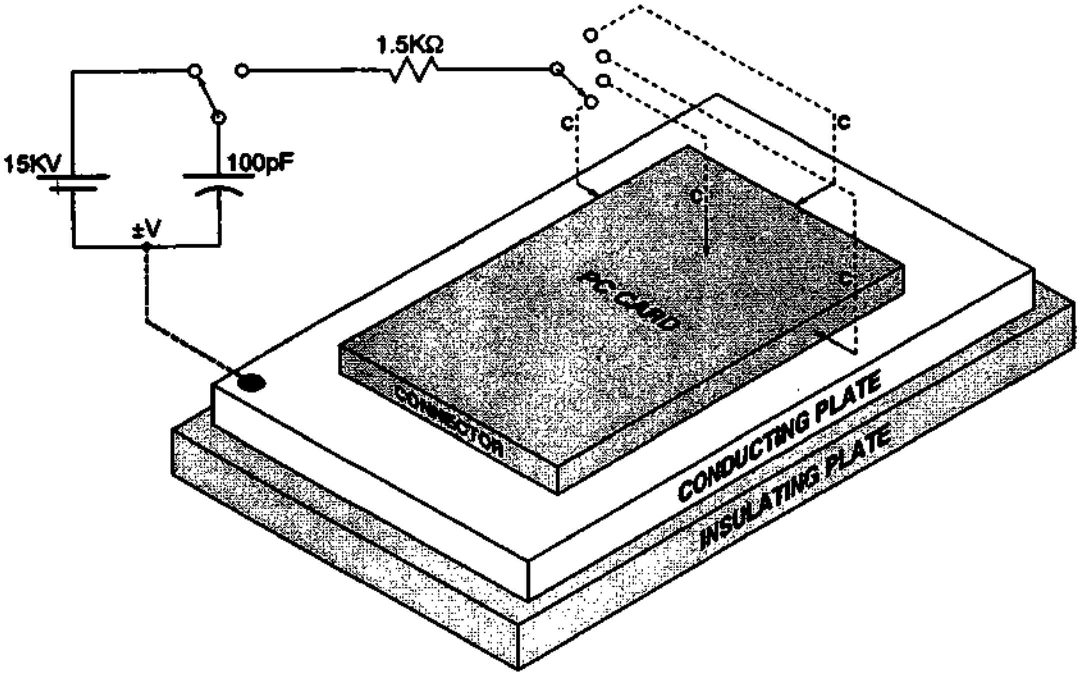


Figure 3-16. Connector Shock & Vibration Test Fixture



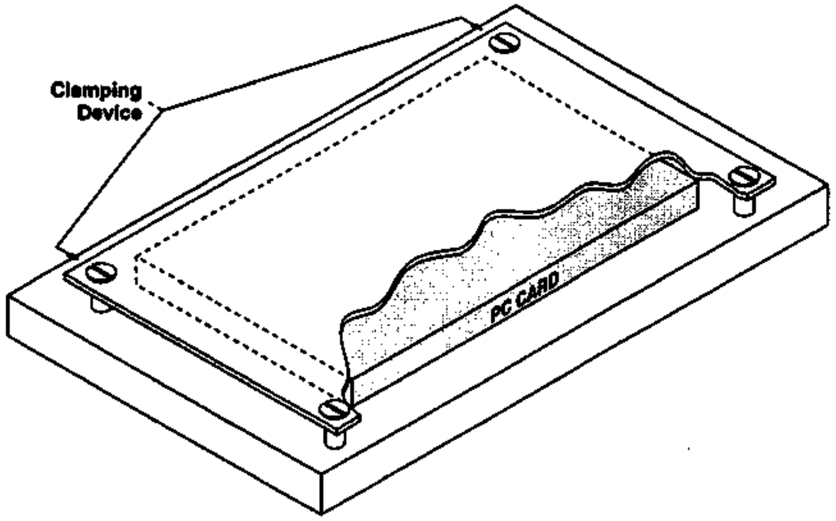
△1 Connect to the four (4) ground contacts (Pin No's 1, 34, 35 and 68)

Figure 3-17. Electrostatic Discharge Test-1 Fixture



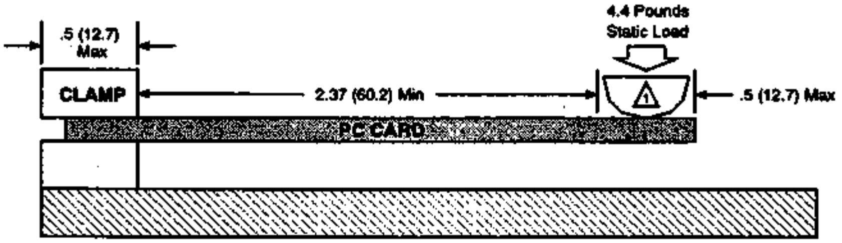
1. PC Card to make contact with the conducting plate. Discharge to top cover, left side, non-connector end and right side three times each. Total discharge cycles = 12 on each side.
2. The pc card cover facing the conducting plate, should make contact with the conducting plate during test.

Figure 3-18. Electrostatic Discharge Test-2 Fixture



1. The pc card shock and vibration test fixture shall entrap the pc card such that all shock and vibration shall be transmitted into

Figure 3-19. PC Card Shock and Vibration Test Fixture

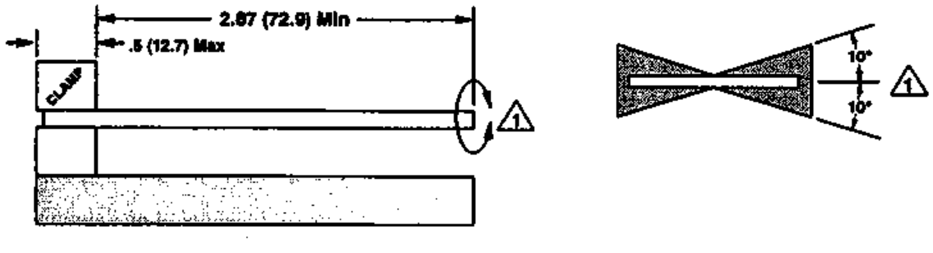


△1 The contact radius force bar is 0.197" ±0.039" (5.0).

2. The force bar shall apply a uniform force across the end of the PC Card.

3. Millimeters are in parenthesis ( ).

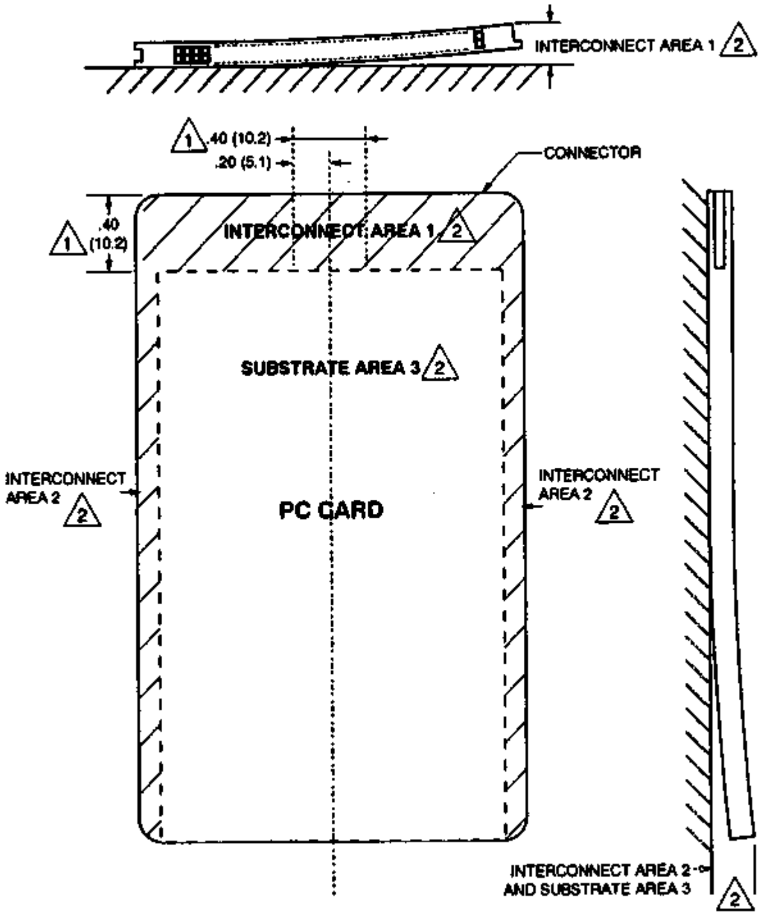
Figure 3-20. Bend Test Fixture



△1 Apply torque to unclamped end of PC card. The torque and angle Max are : Torque 11in-lbs (12.6 kg-cm) or 10° whichever occurs first.

1. Millimeters are in parenthesis ( ).

Figure 3-21. Torque Test Fixture



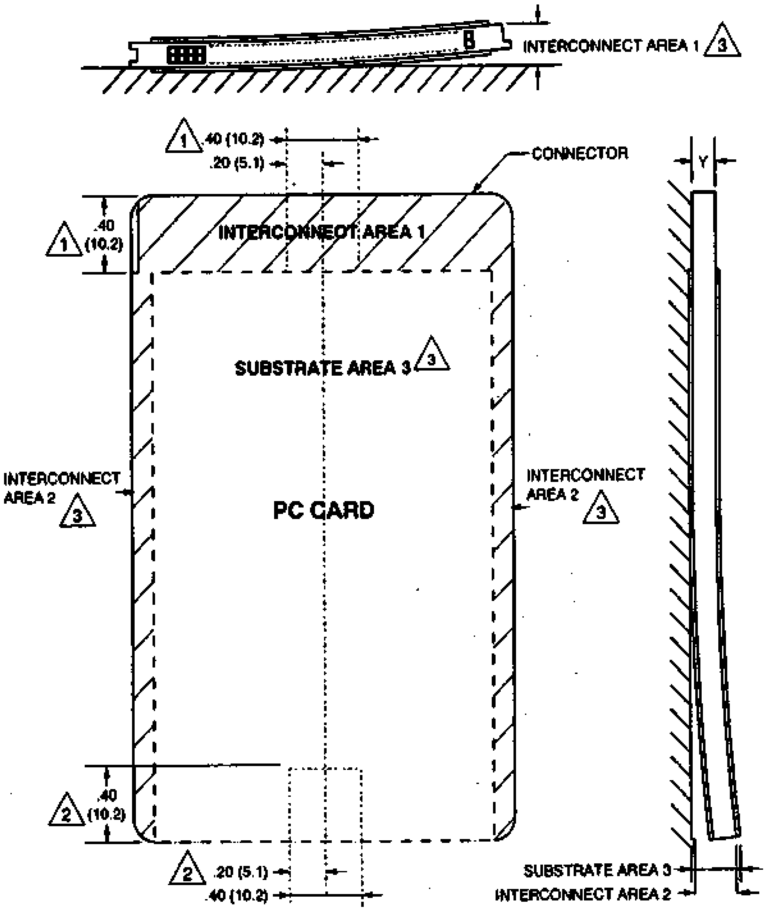
△1 PC CARD THICKNESS (T MIN) MEASUREMENT AREA

△2 WARPAGE IN INTERCONNECT AREA 1 AND 2
AND SUBSTRATE AREA 3 ARE DEFINED AS:

AREA	WARPAGE SPECIFICATION
INTERCONNECT AREA 1	Tmax ≤ Tmin+0.006 (0.15)
INTERCONNECT AREA 2	Tmax ≤ Tmin+0.014 (0.35)
SUBSTRATE AREA 3	Tmax ≤ Tmin+0.020 (0.5) ≤ 0.150 (3.8)

3. MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-22. Type I PC Card Warpage Test Fixture



△1 PC CARD INTERCONNECT THICKNESS (T MIN) MEASUREMENT AREA

△2 PC CARD SUBSTRATE THICKNESS (T SUB) MEASUREMENT AREA

△3 WARPAGE IN INTERCONNECT AREA 1 AND 2 AND SUBSTRATE AREA 3 ARE DEFINED AS:

AREA	WARPAGE SPECIFICATION
INTERCONNECT AREA 1	Tmax ≤ Tmin+0.006 (0.15)
INTERCONNECT AREA 2	Tmax ≤ Tmin+0.014 (0.35)
SUBSTRATE AREA 3	Tmax ≤ Tsub+0.020 (0.5) ≤ 0.211 (5.35)

4 THE PC CARD SUBSTRATE AREA 3 THICKNESS SHALL NOT EXCEED: T SUB±0.008 (0.2)

5. MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 3-23. Type II PC Card Warpage Test Fixture

SECTION - 4

CARD INTERFACE

CARD INTERFACE

4.1 Pin Assignments[1]

Table 4-1: PCMCIA PC Card Pin 1 To Pin 34 Assignments

Memory Only Card Interface: (Always available at card insertion[1])						I/O and Memory Card Interface: (Available only after card and socket are configured)				
Pin	Signal	I/O	Function	+/-	Notes	Pin	Signal	I/O	Function	+/-
1	GND		Ground			1	GND		Ground	
2	D3	I/O	Data bit 3			2	D3	I/O	Data bit 3	
3	D4	I/O	Data bit 4			3	D4	I/O	Data bit 4	
4	D5	I/O	Data bit 5			4	D5	I/O	Data bit 5	
5	D6	I/O	Data bit 6			5	D6	I/O	Data bit 6	
6	D7	I/O	Data bit 7			6	D7	I/O	Data bit 7	
7	CE1	I	Card enable	-	3	7	CE1	I	Card enable	-
8	A10	I	Address bit 10			8	A10	I	Address bit 10	
9	OE	I	Output enable	-		9	OE	I	Output enable	-
10	A11	I	Address bit 11			10	A11	I	Address bit 11	
11	A9	I	Address bit 9			11	A9	I	Address bit 9	
12	A8	I	Address bit 8			12	A8	I	Address bit 8	
13	A13	I	Address bit 13			13	A13	I	Address bit 13	
14	A14	I	Address bit 14			14	A14	I	Address bit 14	
15	WE/PGM	I	Write enable	-		15	WE/PGM	I	Write enable	-
16	RDY/BSY	O	Ready/busy	+/-	2, 4	16	IREQ	O	Interrupt Request	-
17	Vcc					17	Vcc			
18	Vpp1		Programming Supply Voltage 1		2, 3	18	Vpp1		Programming and Peripheral Supply	
19	A16	I	Address bit 16			19	A16	I	Address bit 16	
20	A15	I	Address bit 15			20	A15	I	Address bit 15	
21	A12	I	Address bit 12			21	A12	I	Address bit 12	
22	A7	I	Address bit 7			22	A7	I	Address bit 7	
23	A6	I	Address bit 6			23	A6	I	Address bit 6	
24	A5	I	Address bit 5			24	A5	I	Address bit 5	
25	A4	I	Address bit 4			25	A4	I	Address bit 4	
26	A3	I	Address bit 3			26	A3	I	Address bit 3	
27	A2	I	Address bit 2			27	A2	I	Address bit 2	
28	A1	I	Address bit 1			28	A1	I	Address bit 1	
29	A0	I	Address bit 0			29	A0	I	Address bit 0	
30	D0	I/O	Data bit 0			30	D0	I/O	Data bit 0	
31	D1	I/O	Data bit 1			31	D1	I/O	Data bit 1	
32	D2	I/O	Data bit 2			32	D2	I/O	Data bit 2	
33	WP	O	Write protect	+	2, 3	33	IOIS16	O	IO Port Is 16-bit	-
34	GND		Ground			34	GND		Ground	

NOTES: Active "low" signals are indicated by -(minus). Active "high" signals are indicated by +(plus).

1. Wait and Reset are RFU (no connect) in *PCMCIA PC Card Standard*, Release 1.0. Both must be implemented in the system for compliance with *PCMCIA PC Card Standard*, Release 2.

2. Use of signal changes between memory only and I/O interface.

3. Signal must not be connected between cards. If it is an output signal from the card, it must not be directly connected to any signal source within the host. It must not be wire-OR'd or wire-AND'd with any host signals.

[1] The glossary in Appendix A defines many of the technical terms in this chapter.

Table 4-2: PCMCIA PC Card Pin 35 To Pin 68 Assignments

Memory Only Card Interface (Always available at card insertion)						I/O and Memory Card Interface (Available only after card and socket are configured)				
Pin	Signal	I/O	Function	+/-	Notes	Pin	Signal	I/O	Function	+/-
35	GND		Ground			35	GND		Ground	
36	CD1	O	Card detect	-	3	36	CD1	O	Card detect	-
37	D11	I/O	Data bit 11			37	D11	I/O	Data bit 11	
38	D12	I/O	Data bit 12			38	D12	I/O	Data bit 12	
39	D13	I/O	Data bit 13			39	D13	I/O	Data bit 13	
40	D14	I/O	Data bit 14			40	D14	I/O	Data bit 14	
41	D15	I/O	Data bit 15			41	D15	I/O	Data bit 15	
42	CE2	I	Card enable	-	3	42	CE2	I	Card enable	-
43	RFSH	I	Refresh			43	RFSH	I	Refresh	
44	RFU		Reserved		2	44	IORD	I	IO Read	-
45	RFU		Reserved		2	45	IOWR	I	IO Write	-
46	A17	I	Address bit 17			46	A17	I	Address bit 17	
47	A18	I	Address bit 18			47	A18	I	Address bit 18	
48	A19	I	Address bit 19			48	A19	I	Address bit 19	
49	A20	I	Address bit 20			49	A20	I	Address bit 20	
50	A21	I	Address bit 21			50	A21	I	Address bit 21	
51	Vcc					51	Vcc			
52	Vpp2		Programming Supply Voltage 2		2, 3	52	Vpp2		Programming and Peripheral Supply 2	
53	A22	I	Address bit 22			53	A22	I	Address bit 22	
54	A23	I	Address bit 23			54	A23	I	Address bit 23	
55	A24	I	Address bit 24			55	A24	I	Address bit 24	
56	A25	I	Address bit 25			56	A25	I	Address bit 25	
57	RFU		Reserved			57	RFU		Reserved	
58	RESET	I	Card Reset	+	1, 5	58	RESET	I	Card Reset	+
59	WAIT	O	Extend bus cycle	-	1, 3	59	WAIT	O	Extend bus cycle	-
60	RFU		Reserved		2, 3	60	INPACK	O	Input Port Acknowledge	-
61	REG	I	Register select	-	2,	61	REG	I	Register select & IO Enable	-
62	BVD2	O	Battery voltage detect 2		2, 3	62	SPKR	O	Audio Digital Waveform	-
63	BVD1	O	Battery voltage detect 1		2, 3	63	STSCHG	O	Card Statuses Changed	-
64	D8	I/O	Data bit 8			64	D8	I/O	Data bit 8	
65	D9	I/O	Data bit 9			65	D9	I/O	Data bit 9	
66	D10	I/O	Data bit 10			66	D10	I/O	Data bit 10	
67	CD2	O	Card detect	-	3	67	CD2	O	Card detect	-
68	GND		Ground			68	GND		Ground	

NOTES: I - Input to card, O - Output from card, I/O - Bidirectional

4. Signal must not be connected between cards when I/O interface is supported. If it is an output signal from the card, it must not be directly connected to any signal source within the host. It must not be wire-OR'd or wire-AND'd with any host signals
5. Reset must not be connected between cards unless all cards are reset when any card has Vcc power removed.
6. In systems which switch Vcc individually to cards, no signal should be directly connected between cards other than ground.

4.2 Memory Card Features

Table 4-3: Features of PCMCIA Memory Card

Item	Feature
Access	Random access
Data Bus	Bus 16 bits/8 bits
Memory Types	MaskROM, OTPROM, EPROM, EEPROM, Flash-Memory, SRAM
Memory Capacity	64MB (A0-A25) maximum
REG function	Attribute Memory for storing card identification
I/O Address Space	64 Mbyte (A0-A25) maximum (64 Kbyte for PC compatible architectures)
I/O Space Decoding	Overlapping I/O Address Window: card performs partial selection decoding Independent I/O Address Window: system performs entire selection decoding
I/O Interrupts	1 Interrupt Request signal per card. Routed to specific interrupt level by the host system

4.2.1 Memory Types and Speed Version

Table 4-4: Memory Types and Speed Version

	Speed Version				
Memory Type	600ns[1]	250ns	200ns	150ns	100ns
SRAM	Defined	defined	defined	defined	defined
MaskROM, OTPROM, EPROM, EEPROM, Flash-Memory	Defined	defined	defined	defined	defined

[1] Specified for Dual Operating Voltage Cards while operating at Reduced Operating Voltage (Vcc = 3.3 volts).

4.2.2 Memory Address Space

A separate memory address space of 64 megabytes (A0-A25) is permitted for each memory card installed in a system. The Common Memory may be accessed by a host system for memory read and write operations. Direct-memory access (DMA) read and write operations to Common Memory are possible when Common Memory is mapped directly into a DMA controller's address space.

There is an additional 64 megabyte address space for Attribute Memory which is selected by the -REG signal at the interface. This memory space may not be used for DMA operations.

The Attribute Memory space may be divided into areas for:

1. Card Information Structure — a description of the card's capabilities and specifications and (optionally) its use,
2. Configuration Registers — an optional set of registers which allow the card to be configured by the system, and
3. Reserved Area— the portion of the Attribute Memory space which has not yet been specified.

The size of each of these areas is determined by the card vendor. The Card Information Structure must begin at address 0 but need not be a single, contiguous region. It is recommended that the Card Information Structure and the Card Configuration Registers be located at relatively low addresses to ensure their accessibility by all hosts.

4.2.3 Memory Only Interface

The Memory-Only Interface supports memory cards, but does not contain signals which support I/O Cards. The signals +RDY/-BSY, WP, BVD1 and BVD2 are present on the Memory-Only Interface but replaced by other signals when the I/O Interface is selected. Cards and systems which are designed to *PCMCIA PC Card Standard,* Release 1.0, do not support the RESET or -WAIT signals.

The Memory-Only Interface is the default selected in both the socket and the card whenever a card is inserted into a socket, and immediately following the application of Vcc or the RESET signal to a card. This interface is required to be implemented in all Release 2 compliant systems.

After a Card's Card Information Structure has been interpreted, the card and the socket may be configured, if appropriate, to use the I/O Interface (described in Section 4.2.5).

4.2.4 I/O Address Space

The hardware interface supports a single I/O address space of 64 megabytes (A0 to A25) for peripheral-device access. The I/O address space is shared and divided among all the cards installed in the system. However, many system architectures (such as the 80x86 architectures found in many PCs) support only a 64 kilobyte I/O address space.

I/O registers (ports) may be 8 or 16 bits wide. An -IOIS16 signal is activated by each I/O card when the address at the interface corresponds to a 16-bit I/O register. This permits hardware in a system to adjust the access width (8 or 16 bits) to match the size of I/O port being addressed. When a 16-bit operation is attempted to an 8-bit I/O port, the system hardware may divide the operation into two consecutive 8-bit operations as is done in PC systems that support the ISA bus structure.

Peripheral cards may be designed such that the host system alone determines when the card is selected. Alternatively, both the host and card may play a role in determining when the latter is selected. Thp card includes information in the Card Information Structure which tells the host the address decodings the card may be configured to perform. The host then programs the card to perform a particular decoding using the card's Configuration Registers.

To ensure compatibility in peripheral cards which completely emulate existing fixed-address peripherals, the cards may decode a portion of the address space. For example, a card might decode only A0 through A9 and respond only when a range of addresses corresponding to the peripheral's registers are selected. Correspondingly, the system could decode address lines A9 and A8 and simultaneously select all the appropriately configured peripheral cards only when A9 or A8 is high. A peripheral card drives the Input Port Acknowledge signal (-INPACK) low when an input port on the card is being accessed. This allows any data buffers in the host between the card and the host's internal bus to be activated during the access.

Peripheral cards must be configured by the system before their I/O address space becomes accessible. It is recommended that new devices which do not require software compatibility with existing drivers decode only enough address lines to address the number of I/O ports implemented. Furthermore, they should allow the system to locate them arbitrarily in the I/O address space, thereby eliminating conflicts with other I/O devices.

4.2.5 I/O Interface

The I/O Interface requires that the Memory-Only Interface also be implemented within the same socket, and that the Memory-Only Interface be selected in the socket when no card is inserted and immediately following Card reset and the application of Vcc to the card. The I/O interface contains all the signals in the Memory- Only Interface with the exception of the +RDY/-BSY, WP, BVD1 and BVD2 signals.

The I/O interface also supports the following additional signals, some of which replace Memory Only signals not supported on the I/O interface:

> Interrupt Request (-IREQ), I/O Port is 16 bits (-IOIS16), I/O Read strobe (-IORD), I/O Write strobe (-IOWR), Input Port Acknowledge (-INPACK), audio digital waveform intended for a speaker (-SPKR), and a card Status Changed (-STSCHG) signal.

The Extend Bus Cycle (-WAIT) and Card Reset (+RESET) signals, which are not required in a *PCMCIA PC Card Standard*, Release 1.0 memory interface, are required for both the Memory-Only and the I/O-Card interfaces in all systems supporting Release 2.

Peripheral cards must be configured by the system before their I/O Interface becomes active. Before configuring a card, the system must examine the card's Card Information Structure to determine the I/O address space, interrupt request, and other requirements of the possible card configurations. The system uses this information to select the best configuration from those available in the card, as determined by the system's hardware and software capabilities, as well as the requirements of other cards installed concurrently. If no card configuration is suitable for the system, because the card requires resources which are not available in the system, or which have already been assigned by the system to other cards, the system may reject the card without configuring it.

Unless otherwise specified, all signals must be implemented by the system to be compliant with I/O portion of the standard. Since, systems with 8-bit data buses are not required to implement 16-bit data buses or 16-bit operations, they must keep the -CE2 signal in the inactive state at all times.

4.2.6 Custom Interfaces

Systems may provide custom interfaces through a standard socket. Custom interfaces are expected to support enhanced features, such as internal-bus extensions, or customized signals not applicable across architectures. A card or a socket may support more than one custom interface.

A custom interface is handled by the system and the card in a manner similar to an I/O interface. Both the socket and the card must use a Memory-Only Interface when a card is inserted, or when power is removed from a card. After a card is powered up, the system reads the card's Card Information Structure. If the card is found to support a custom interface also supported by the host socket, the card and the socket may be configured to the custom-interface mode. The card is configured using its Configuration Index Register.

Refer to Section 4.15 for information on configurable cards and to Section 5.1.3 for information on Card Information Structure data relating to custom interfaces.

4.2.7 Configurable Cards

Certain memory and all peripheral cards may be configured by the system. This is to ensure that cards incompatible with the system, or with other cards installed in the system, are either compatibly reconfigured, or rejected, if a compatible configuration is not available on the card.

The system adjusts the card using the Card Configuration Registers. These are located in the card's Attribute-Memory space, at the location indicated in the card's Card Information Structure.

Refer to Section 4.14 for information on configurable cards.

4.2.8 Compatibility Between Revision Levels of this Standard

PCMCIA PC Card Standard, Release 1.0 defined the Memory-Only interface without a Card Reset (RESET) input signal and Extended Bus Cycle (WAIT) signal. Systems built to the Release 1.0 interface have the option of leaving the RESET pin as no connect or (card) Vcc and the WAIT pin as a no connect. When a Release 2 Memory Card is inserted into a Release 1.0 socket the +Reset signal will appear asserted continuously. In order to be backward compatible a Release 2.01 Memory Card inserted into a Release 1.0 socket must appear to the host system on Power-on, after the 20 millisecond Vcc settling time, as a Release 1.0 compliant card in its initial default power-on state.

Release 2 cards which are not intended for operation in Release 1.0 host-system sockets (e.g. I/O cards or mixed I/O-memory cards) need not present a valid CIS while RESET is asserted, and must not reply to read commands or act on write commands while RESET is asserted.

A Release 2 host system socket, upon recognizing a Release 2 Memory Card, can subsequently take advantage of the RESET and WAIT signals employed on the card.

4.3 Signal Description

Signals on the PCMCIA interface, whether operating at the standard operating supply level of 5 volts or the reduced operating supply voltage of 3.3V, are considered asserted within the range of V_{OH} and negated within the range of V_{OL}. However, the V_{OH} and V_{OL} ranges are determined by the supply voltage as shown in the following table. V_{OH}, V_{IH}, V_{OL} and V_{IL} for 3.3V operation are under review in JEDEC, and the resulting JEDEC standard will be adopted by this standard. All signals are considered to be active when the line is asserted,(+), unless the signal name is preceded by the minus sign, (-), in which case it shall be considered active when the line is negated.

Table 4-5: PC Card Logic Levels

DC Levels		Host		Card
Parameter	Conditions	Min	Max	
V_{IH}	Vcc = 5 V +/- 5%	2.4 V	Vcc+.25 V	TTL or CMOS
V_{IL}	Vcc = 5 V +/- 5%	0.0 V*	0.8 V	TTL or CMOS
V_{OH}	Vcc = 5 V +/- 5%	2.8V (.9Vcc)[1]	Vcc	TTL or CMOS
V_{OL}	Vcc = 5 V +/- 5%	0.0 V	0.5 V (.1 Vcc)[1]	TTL or CMOS
V_{IH}	Vcc = 3.3V +/- 5%	Pending JEDEC Review		
V_{IL}	Vcc = 3.3V +/- 5%	Pending JEDEC Review		
V_{OH}	Vcc = 3.3V +/- 5%	Pending JEDEC Review		
V_{OL}	Vcc = 3.3V +/- 5%	Pending JEDEC Review		

[1] (for CMOS Loads)

All signals are grouped under four classifications: I (Input), O (Output), I/O (Bidirectional), and R (Reserved). Input signals are those driven by the host and output signals are those driven by the PC Card.

All pins identified as ground shall be connected to signal ground at the host. Signal pins identified as *reserved* shall have no connection at either the host or the card.

The data path to the memory card is 16 bits wide and consists of signals D0-D15. The card supports a 26-bit address bus, (A0-A25), for a maximum 64 megabyte addressing range.

4.3.1 Address BUS (A0-A25)

Signals A0 through A25 are address-bus-input lines which enable direct address of up to 64 megabytes of memory on the card. Signal A0 is not used in word-access mode. Signal A25 is the most significant bit, and bit number (and significance) decrease downward to A0.

4.3.2 Data BUS (D0-D15)

Signals D0 through D15 constitute the bidirectional data bus. The most significant bit is D15. Bit number (and significance) decrease downward to D0.

4.3.3 Card Enable (-CE1 & -CE2)

The -CE1 and -CE2 lines are active-low, card-enable, input signals. The -CE1 input enables evennumbered-address bytes and -CE2 enables odd-numbered-address bytes. A multiplexing scheme based on A0, -CE1 and -CE2 allows 8-bit hosts to access all data on D0 through D7 if desired. See Table 4-7.

To ensure data retention on battery-backed-up SRAM cards, and permit power-up initialization of peripheral cards, a minimum of 20 milliseconds must elapse after:

1. the application of Vcc to the card, or
2. the end of the RESET signal to the card (in systems which support the RESET signal),

whichever event occurs latest.

The card enables are used to access both Common and Attribute Memory, and to access I/O.

Refer to Section 4.7 for additional information on Common Memory Read Functionality.

Refer to Section 4.14 for additional information regarding I/O Read and Write Functionality.

4.3.4 Output Enable (-OE)

The -OE line is the active-low, input signal used to gate Memory Read data from the memory card. SRAM memory cards fall into two categories:

1. cards where the -OE signal must be negated during write operations, and
2. cards that do not use the -OE signal during write operations and allow the signal to be in either state.

Hosts must negate the -OE signal during write operations.

4.3.5 Write Enable/Program (-WE/-PGM)

The -WE/-PGM input signal is used for strobing Memory Write data into the memory card. This line is also used for memory cards employing programmable-memory technologies. See Section 5.2.6 for identification of Programmable Memory Technology cards.

4.3.6 Ready/Busy (+RDY/-BSY)

The Ready/Busy function is provided by the +RDY/-BSY signal when the card and the host socket are configured for the Memory-Only Interface. When a host socket and the card inserted into it are both configured for the I/O interface, the +RDY/-BSY function is provided by the +RDY/-BSY status bit in the card's Pin Replacement Register. When the Pin Replacement Register is not implemented on a card configured for the I/O Interface, the +RDY/-BSY function is continuously in the Ready condition. The following descriptions of the +RDY/-BSY signal apply equally to both the +RDY/-BSY signal of the Memory- Only Interface and to the +RDY/-BSY bit in the Pin Replacement Register of a card configured for the I/O Interface. See Section 4.14.3 for a description of the Pin Replacement Register.

The +RDY/-BSY signal is a status signal polled by the host, or used by the socket to generate an interrupt on the Busy to Ready transition or on both transitions. It is intended to indicate the completion of potentially lengthy operations within the a card. It is not intended to delay the completion of a machine cycle at the PCMCIA interface or on the Host's internal bus, the -WAIT signal is available for that purpose.

The +RDY/-BSY line is driven low by the PC Card to indicate that the PC Card circuits are busy and unable to accept some data-transfer operations. The +RDY/-BSY signal is set low while the card is busy processing a previous command or performing initialization. The signal +RDY/-BSY is set high when the PC Card is ready to accept a new data-transfer command.

When independent, unrelated sources for the +RDY/-BSY signal are present on the card, +RDY/-BSY signal shall by Busy while any of the unrelated sources of the +RDY/-BSY on the card require the blocking of certain host accesses and are indicating Busy. A group of related devices on the card and managed by the same host software shall be treated collectively as a single independent source of +RDY/-BSY.

A PC Card is permitted to process subsequent operations from the host while the +RDY/-BSY signal is low if its internal circuits allow proper operation. It is the responsibility of the card or device vendor to communicate (through CIS descriptions or other means) those operations which are permitted to a card while the +RDY/-BSY signal is asserted. In the absence of any such knowledge, the host shall not attempt any access to the card while the card is Busy.

The host must not access a card until a minimum of 20 milliseconds has passed after Vcc is stable, and, for Release 2.0 and later systems, after the RESET signal is negated. In addition, the card's +RDY/-BSY line must be high (RDY) before the initial access. If the card will not be initialized and ready for operations at the end of the 20 millisecond waiting period, the +RDY/-BSY signal will be set low (BSY) within 10 microseconds of reset or application of Vcc to the card. A card that requires more than 20 milliseconds for internal initialization before access shall drive +RDY/-BSY to a low (BSY) condition until it is ready for initial access.

Release 1.0 required that cards containing battery-backed-up SRAM not be accessed before 20 milliseconds after stable power is applied. However, some systems otherwise conforming to Release 1.0 may access a card before the end of the initialization period. Therefore, it is recommended that during the initialization period the card's Card Information Structure contain the correct card description. If that is not possible, then it should contain a valid CIS description of a null or ROM device. This will prevent the Release 1.0 system from presuming that the card is an unitialized SRAM card.

[Note that a Release 1.0 system will never remove the reset condition (as +Reset is not connected in a Release 1.0 system).]

If the card will require more than 10 microseconds to enter the Sleep Mode, or to return to operating condition following Card Wakeup, the +RDY/-BSY signal will be set low (BSY) within 10 microseconds after the power-down bit in the Card Configuration and Status Register is changed. Following Card Wakeup the host must not access the card until a minimum of 10 microseconds has passed and the Card's +RDY/-BSY line is in a high (RDY) state. If a card requires more than 10 microseconds following Wakeup, and before the card is ready for operation, the card shall hold the +RDY/-BSY in a low (BSY) state until the card is ready for operation.

When a card and its socket have been configured for the I/O Interface, the +RDY/-BSY status may be available in the Pin Replacement Register and the signal is replaced on interface pin 16 with the -IREQ (Interrupt Request) signal. See Section 4.6.2 regarding the Interrupt Request function.

4.3.7 Card Detect (-CD1 & -CD2)

The -CD1 and -CD2 signals provide for proper detection of PC Card insertion. The signal pins are at opposite ends of the connector to ensure a valid detection (i.e. ensuring both sides of the card are firmly inserted). The -CD1 and -CD2 signals are connected to ground internally on the PC Card and will be forced low whenever a card is placed in a host socket. The host socket interface circuitry shall provide 10K or larger pull-up resistors to Vcc on each of these signal pins.

4.3.8 Write Protect (+WP)

The WP output signal is used to reflect the status of the card's Write Protect switch. If the Write Protect switch is present, this +WP signal will be asserted by the card when the switch is enabled, and negated when the switch is disabled. If the memory card has no Write Protect switch, the card will connect this line to ground or Vcc depending on the condition of the card memory. For example, if the card can always be written to, the pin will be connected to ground, and if the card is permanently Write Protected, the pin will be connected to Vcc.

When a card or socket is configured for the I/O Interface, the Write Protect status may be available in the Pin Replacement Register, and the signal is replaced in the interface by the I/O Port is 16 bits (-IOIS16) signal. Refer to Section 4.6.1 for information on the -IOIS16 signal.

4.3.9 Attribute Memory Select (-REG)

The -REG signal is kept inactive (high) for all Common Memory access. When this signal is active (low), access is limited to Attribute Memory (-OE or -WE active) and to the I/O space (-IORD or -IOWR active). Attribute Memory is a separately-accessed section of card memory and is generally used to record card capacity and other configuration and attribute information. Attribute Memory is also used to access standardized Card Configuration Registers.

The timing of Attribute Memory may be different than that of Common Memory. Refer to manufacturer's specifications for details. When Attribute Memory is accessed, only data signals D0-D7 are valid and signals D8-D15 shall be ignored. Signals -CE1, -CE2 and A0 are still valid, but it is only possible to select even-numbered addresses. A combination of signals -CE1/-CE2/A0 that requests an odd-numbered byte will result in invalid data on the bus. See Table 4-6.

I/O space is used for access to peripheral devices via the -IORD and -IOWR strobe signals. Systems which generate the -IORD and -IOWR strobe signals during DMA operations must keep -REG inactive (i.e. high) during DMA transfers to prevent spurious I/O accesses while a memory address is present on the address lines.

4.3.10 Battery Voltage Detect (BVD1 & BVD2)

The signals BVD1 and BVD2 are generated by the memory card as an indication of the condition of its battery.

Both signals are kept asserted when the battery is in good condition. A replacement warning condition is signalled by BVD1 asserted and BVD2 negated, although data integrity on the card is still assured. If BVD1 is negated, with BVD2 either asserted or negated, the battery is no longer serviceable and data is lost. Refer to Table 4-15.

When the I/O Interface is selected the BVD1 and BVD2 signals are replaced at the interface by the card Status Changed (-STSCHG) and the Audio Digital Waveform (-SPKR) signals respectively. The battery voltage status information may be available in the card's Pin Replacement Register while the I/O Interface is configured. Refer to Sections 4.6.3, 4.6.4 and 4.15.3.

4.3.11 Program and Peripheral Voltages (Vpp1 & Vpp2)

The Vpp1 and Vpp2 signals supply programming voltages for programmable-memory operation, or additional supply voltages for Peripheral Cards. These pins are to be connected to the Vcc voltage level until the Card Information Structure (CIS) of the card has been read and other permissible values for Vpp1 or Vpp2 have been determined. Vpp voltages are used on the card for Peripheral Card operation and for altering programmable memory on the card. The voltage applied to the Vpp pins of a card must never be greater than the Vpp level appropriate for the card. If the appropriate Vpp voltage for a card cannot be determined, the voltage applied to the Vpp pins must not exceed Vcc. Refer to Section 5.3.2.1 and the Card Information Structure for more information on the characteristics of Vpp1 and Vpp2.

Systems are required to be able to supply the Vcc level on the Vpp pins.

It is recommended that systems be able to supply at least the Vpp voltage of +12 Volts ±5% in addition to the Vcc level.

For low power applications it is recommended that the system be able to apply a low logic level to Vpp.

When the Vpp value required by a card is unavailable in a system, the system may reject the card.

4.3.12 Card Voltage and Ground (Vcc & GND)

The Vcc and GND input pins have been placed at symmetrical positions on the memory card to provide safety in the case of an inverted-card insertion. Two power pins (#'s 17 and 51) and four ground pins (#'s 1, 34, 35 and 68) are employed to reduce the impedance between the memory card and the system.

In order to determine a card's characteristics, Vcc must be within the Operating Voltage range (e.g. 5 Volts) when a card is read initially. If the Card Information Structure indicates that the card is a Dual Operating Voltage Card capable of operating at the Reduced Operating Voltage (3.3 Volts), then the Vcc supply voltage may be lowered to the Reduced Operating Voltage, and access timing adjusted according to the indicated information.

4.3.13 Refresh (RFSH)

The Refresh signal is intended for pseudostatic SRAMS (PSRAM). Its use will be more clearly defined in a future version of this standard.

4.3.14 Reserved Pins (RFU)

Several pins have been identified as Reserved for Future Use (RFU). Neither memory cards nor host systems shall make any electrical connections to these pins.

4.4 Release 2 Signals Affecting Both Memory Only and I/O Interfaces

4.4.1 Card Reset (RESET)

The +RESET signal clears the Card Configuration Option Register thus placing a card in an unconfigured (Memory-Only Interface) state. It also signals the beginning of any additional card initialization. The system must place the +RESET signal in high-impedance during card power-up (including both Vcc turn-on and hot insertion). The signal must remain in high impedance for at least 1 ms after Vcc becomes valid. All configurable cards (including all I/O Cards) must monitor +RESET and return to the unconfigured state when +RESET is active. A card remains in the unconfigured state until the Card Configuration Option Register has been written with a valid configuration.

Cards requiring RESET must enter the unconfigured state each time power is applied. For example:

1. the card may generate a power-on RESET internally, or
2. the +RESET signal may be pulled up to Vcc through a >100K resistor on cards requiring reset.

This ensures when a card is inserted into a socket it is reset before the signal pins make contact with the socket, and the card is reset (continuously) when placed into a Release 1.0 compatible socket.

All new system designs — including those which support just the Memory-Only Interface — should support the +RESET and -WAIT function.

System compatibility with Release 2.0 and above requires support of the +RESET signal.

4.4.2 Extend Bus Cycle (-WAIT)

The -WAIT signal is asserted by a card to delay completion of the memory-access or I/O- access cycle then in progress.

All new system designs — including designs which support just the Memory-Only Interface — should support the -WAIT and +RESET function.

System compatibility with Release 2.0 and above requires support of the -WAIT signal.

4.5 I/O Interface Signals Replacing RFU Pins

4.5.1 I/O Read (-IORD)

The -IORD signal is made active to read data from the card's I/O space. The -REG signal and at least one of -CE1 or -CE2 must also be active for the I/O transfer to take place. A PC Card will not respond to the -IORD signal until it has been configured for I/O operation by the system.

4.5.2 I/O Write (-IOWR)

The -IOWR signal is made active to write data to the card's I/O space. The -REG signal and at least one of -CE1 or -CE2 must also be active for the I/O transfer to take place. A PC Card will not respond to the -IOWR signal until it has been configured for I/O operation by the system.

4.5.3 Input Acknowledge (-INPACK) [I/O Operation]

The Input Acknowledge output signal is asserted when the card is selected and the card can respond to an I/O read cycle at the address on the address bus. This signal is used by the host to control the enable of any input data buffer between the card and the CPU. This signal must be inactive until the card is configured.

[Note: In cases where a card is configured to respond to I/O read cycles at all addresses, the -INPACK signal may be asserted whenever the Card Enable (-CE1 and-CE2) inputs are true.]

4.6 I/O Interface Signals Replacing Memory Interface Signals

4.6.1 I/O IS 16 Bit Port (-IOIS16) [replaces WP]

The -IOIS16 output signal is asserted when the address at the socket corresponds to an I/O address to which the card responds, and the I/O Port being addressed is capable of 16-bit access.

When this signal is not asserted during a 16-bit I/O access, the system will generate 8-bit references to the even and odd byte of the 16-bit port being accessed.

4.6.2 Interrupt Request (-IREQ) [replaces +RDY/-BSY]

The Interrupt Request signal is available only when the card and the interface are configured for the I/O and Memory Interface (indicated as Interface Type 1 in the TPCE_IF field of the TPCE tuple, see Section 5.2.8.3). Interrupt Request is asserted by an I/O Card to indicate to the host that a card device requires host software service. The interrupt signal at the interface is routed by the system to one of the interrupt request signals on the system's internal bus. The signal is held at the inactive level when no interrupt is requested.

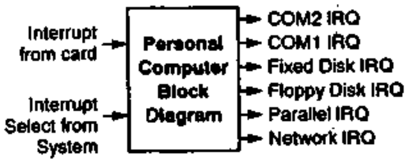
4.6.2.1 Interrupt Request Routing

A general-purpose system should be able to route each card's interrupt request to any of the interrupt-request levels used for installable devices within the system. A system which is both hardware- and software-dedicated to a particular application may support only the subset of interrupt-request levels necessary for the application. Driver software customization will be necessary to support I/O cards in the dedicated-system environment.

Implementation Note:

For "PC" compatible computers it is recommended that -IREQ from the card be able to be routed to at least the following interrupt signals in the system.

Function	AT IRQ	XT IRQ
Communications 2 (COM2)	IRQ3	IRQ3
Communications 1 (COM1)	IRQ4	IRQ4
Fixed Disk	IRQ14	IRQ5
Floppy Disk	IRQ6	IRQ6
Parallel Port (LPT1)	IRQ7	IRQ7
Network/Other	IRQ10	Not Available



4.6.2.2 Pulsed- and Level-Mode Interrupt Support

The Interrupt Request may be either a pulse or a level depending upon the needs of the system.

I/O cards designed to operate in a variety of machines should support both level-mode and pulsed-mode interrupts. Therefore, it is recommended that I/O Cards support both of these modes.

The host system selects the mode of interrupt — level or pulsed — through the Card Configuration Registers. Refer to Section 4.14.2.

4.6.2.3 Pulsed-Mode Interrupt Signal

A pulsed-mode interrupt is asserted by placing an active- going (i.e. low) pulse on the interrupt line. Pulsed-mode interrupts are generally utilized by systems using the "ISA" PC architecture in which interrupts are edge sensitive. The pulse width must be at least 0.5 microseconds, and the interrupt will be recognized by the host on the trailing (i.e. rising) edge of the pulse.

[Note: The pulsed-mode interrupt may be lost when more than one interrupting device shares an interrupt-request signal at the system bus, and standard system and application software is used. Interrupt requests may be lost if they arrive at the system before a prior request on the shared interrupt-request signal has been fully serviced.]

4.6.2.4 Level-Mode Interrupt Signal

A level-mode interrupt is asserted by placing the Interrupt Request signal in an active (i.e. low) state until the interrupt has been serviced by the system. The interrupt-request signal is then held in the inactive state. The level-mode interrupt is standard in PC- compatible systems using the Micro Channel Architecture, as well as in many non-PC-compatible systems.

Cards must support the level-interrupt-request mode.

4.6.2.5 Interrupts and +RDY/-BSY

Pin 16 serves as +RDY/-BSY in memory-only cards, and as -IREQ for I/O cards. Pin 16 is used as +RDY/-BSY while an I/O-capable card is configured for the Memory-Only Interface. See the +RDY/-BSY description in Section 4.3.6.

4.6.3 Audio Digital Waveform (-SPKR) [replaces BVD2]

This Binary Audio signal is an optional signal which may be available only when the card and the socket have been configured for the I/O Interface. It provides a single-amplitude, on-off, (binary) audio waveform intended to drive the host's loudspeaker. The signal to the speaker should be generated by taking the exclusive-OR function of the -SPKR signals from all those cards providing Binary Audio, and from the system speaker source. When no audio signal is present, or if the card does not support the Binary Audio function, the -SPKR signal shall be held inactive (i.e. high).

Cards which supply the Binary Audio function are required to support the Audio Enable bit in the Card Configuration Registers. This allows the system to selectively enable or disable the audio function. Refer to Section 4.15.2.

PCMCIA recommends, but does not require, that systems support the Binary Audio function.

The BVD2 signal is available on the same pin as -SPKR when the Card and Socket are configured for the Memory Only Interface.

4.6.4 Status Changed (-STSCHG) [replaces BVD1]

Status Changed is an optional signal used to alert the system to changes in the Ready/Busy (RDY/BSY), Write Protect (WP), or Battery Voltage (BVD) conditions of the card while the I/O Interface is configured.

The signal is held inactive (i.e. high) if the function is not supported by the card or when the "SigChg" bit in the Card Status Register is false (logic 0). When the "SigChg" bit is true (logic 1), the Status Changed signal is active (i.e. low) when the "Changed" bit in the Card Status Register is true (logic 1), and the signal is inactive (i.e. high) when the "Changed" bit is false (logic 0). The Changed bit is the logical OR result of the individual changed bits — CBVD1, CBVD2, CWP, and CRDBSY — in the Pin Replacement Register.

The system mechanism used for BVD1 fail detection may be used to detect a change of status signals. Therefore cards which are configured for I/O operation may continue to notify the system of changes of these status signals although the status signals no longer appear as independent signals on the card interface.

While the card and socket are configured for the Memory-Only Interface, the BVD1 signal is available on this pin. Refer to Section 4.15.3.

4.7 Operating Conditions

Item	Symbol	Conditions
Operating Voltage (All Cards)	Vcc	5V, ±5%
Reduced Operating Voltage (Dual Operating Voltage Cards Only)	Vcc	3.3V, ±5%
Signal Interface Level	-	TTL or CMOS Level

4.7.1 Default Conditions and Card Identification

When a card is first inserted, or when the system cannot determine whether or not the card has been changed, the following procedure must be followed before the Reduced Operating Voltage may be applied to the card.

When power is initially applied to a card, the normal Operating Voltage (5 volts) must be applied to the card. If the system determines from the card's Card Information Structure (CIS) that the card is operable at the Reduced Operating Voltage (3.3 volts), then at the system's option the Reduced Operating Voltage may be applied to the card. While the Reduced Operating Voltage is applied to the card, the card's access timing must be adjusted to meet the criteria indicated in the card's CIS.

If the system cannot determine (from the card's CIS) that the card is capable of operation at the Reduced Operating Voltage, then the card must be accessed at the normal Operating Voltage.

4.8 Memory Function

4.8.1 Common Memory Function

This section describes operations of the Common Memory area.

4.8.2 Common Memory Read Function

The memory card can be configured with different types of memory devices (such as SRAM, MaskROM, etc.), however, the Read function shares common signal-state sequencing.

To access Common Memory, the signal -REG shall be kept inactive, and the signal -OE shall be active during the Read cycle. Signals -CE1 and -CE2 control the activation of the Memory Card and A0 control-byte ordering on the data- bus lines D0-D15. Table 4-6 shows the signal states and data bus validity for the Read functions described below.

When both -CE1 and -CE2 are inactive, the card is in standby mode. When either -CE1 or -CE2 become active (low), the memory card is activated and ready for data transfers. When -CE1 is active and -CE2 is not active, Byte Access mode is enabled (8-bit transfers). Both the even-byte data and odd- byte data outputs will be valid in data bus lines D0-D7. The selection of an even-byte or an odd-byte is controlled by signal A0.

When using word access (16-bit transfers), both -CE1 and -CE2 are active (low), and the even-byte data and odd-byte data outputs are valid in data bus lines D0-D15. During Word mode, signal A0 is ignored.

Odd-Byte-Only access is enabled by -CE1 being inactive and -CE2 active. During Odd-Byte-Only access, only data lines D8-D15 contain valid data, and address signal A0 is ignored.

Table 4-6: Common Memory Read Function for all types of Memory

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	Vpp2	Vpp1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	Vcc*	Vcc*	High-Z	High-Z
Byte Access (8 bits)	H H	H H	L L	L H	L L	H H	Vcc* Vcc*	Vcc* Vcc*	High-Z High-Z	Even-Byte Odd-Byte
Word Access (16 bits)	H	L	L	X	L	H	Vcc*	Vcc*	Odd-Byte	Even-Byte
Odd-Byte Only Access	H	L	H	X	L	H	Vcc*	Vcc*	Odd-Byte	High-Z

* Additional Vpp2 and Vpp1 values are permitted when cards use Vpp voltages as additional power supply levels for other card functions as indicated by Card Information Structure. Refer to Sections 4.14 and 5.2.7.

4.8.3 Common Memory Write Function for SRAM, EEPROM and Single-Supply Flash Cards.

During Write mode, the function of signals -REG, -CE1, -CE2 and A0 are the same as in the Read mode.

During Write mode, signal -OE must be kept inactive, and signal -WE/-PGM is active. The Memory Card can perform Write operations in 3 modes: Byte access, Word access, and Odd-Byte-Only access. Refer to Table 4-7 for signal states and data-bus validity for Common Memory Write modes.

Table 4-7: Common Memory Write Function for SRAM, EEPROM and Single-Supply Flash Cards

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	Vpp2	Vpp1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	Vcc*	Vcc*	XXX	XXX
Byte Access (8 bits)	H H	H H	L L	L H	H H	L L	Vcc* Vcc*	Vcc* Vcc*	XXX XXX	Even-Byte Odd-Byte
Word Access (16 bits)	H	L	L	X	H	L	Vcc*	Vcc*	Odd-Byte	Even-Byte
Odd-Byte-Only Access	H	L	H	X	H	L	Vcc*	Vcc*	Odd-Byte	XXX

* Additional Vpp2 and Vpp1 values are permitted when cards use Vpp voltages as additional power supply for other card functions as indicated by the CIS. Refer to Sections 4.14 and 5.2.7.

4.8.4 Common Memory Write Function for OTPROM, EPROM and Flash-Memory

Memory write functions for programming operations are vendor specific.

4.8.5 Attribute Memory Function

Attribute Memory is an optional space intended for storing memory-card identification and configuration information, and does not require a large address space. Attribute Memory is limited to 8-bit wide access for economical reasons.

4.8.6 Attribute Memory Read Function

For the Attribute Memory Read function, signals -REG and -OE must be active during the cycle. As in the Common Memory Read function, the signals -CE1 and -CE2 control the even-byte and oddbyte address, but only even-byte data is valid during the Attribute Memory Read function. Refer to Table 4-8 for signal states and bus validity for the Attribute Memory Read function.

Table 4-8: Attribute Memory Read Function

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	Vpp2	Vpp1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	Vcc*	Vcc*	High-Z	High-Z
Byte Access (8 bits)	L L	H H	L L	L H	L L	H H	Vcc* Vcc*	Vcc* Vcc*	High-Z High-Z	Even-Byte Not Valid
Byte Access (16 bits)	L	L	L	X	L	H	Vcc*	Vcc*	Not Valid	Even-Byte
Odd-Byte- Only Access	L	L	H	X	L	H	Vcc*	Vcc*	Not Valid	High-Z

* Additional Vpp2 and Vpp1 values for Read Function are permitted when cards use Vpp voltages as additional power supply levels rather than only for programmable memory. However, no voltage level other than Vcc may be applied to the Vpp pins until a card has been identified as supporting alternate Vpp levels by reading its Card Information Structure. Refer to Sections 4.14 and 5.2.7.

4.8.7 Attribute Memory Write Function for SRAM, EEPROM and Single-Supply Flash Cards

While writing to Attribute Memory, signals -REG and -WE/-PGM must be kept active for the entire cycle while the signal -OE is kept inactive for the entire cycle. See Table 4-9 for signal states and bus validity for the Attribute Memory Write function.

Table 4-9: Attribute Memory Function for Single Supply SRAM and EEPROM

Function Mode	-REG	-CE2	-CE1	A0	-OE	-WE	Vpp2	Vpp1	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	Vcc*	Vcc*	XXX	XXX
Byte Access (8 bits)	L L	H H	L L	L H	H H	L L	Vcc* Vcc*	Vcc* Vcc*	XXX XXX	Even-Byte XXX
Byte Access (16 bits)	L	L	L	X	H	L	Vcc*	Vcc*	XXX	Even-Byte
Odd-Byte-Only Access	L	L	H	X	H	L	Vcc*	Vcc*	XXX	XXX

* Additional Vpp2 and Vpp1 values for Write Function are permitted when cards use Vpp voltages as additional power supply levels rather than only for programmable memory. Refer to Sections 4.14 and 5.2.7.

4.8.8 Attribute Memory Write Function for Dual Supply OTPROM, EPROM and Flash Cards

Memory write functions for programming operations are vendor specific.

4.8.9 Write Protect Function

Table 4-10 below, describes the write-protection options and corresponding write-protect (WP) switch and signal states.

Table 4-10: Write Protect Function

Memory Writeability Combinations on Card	Symbol	WP Switch	WP Signal	Minimum Card Information Contents Related to Write Protect
Always Writable	A	None	Low	No WP Information Needed - Memory follows WP signal which is always Low (Not Protected). Optionally the Card info may specify all devices as Always writeable.
Never Writable	N	None	High	No WP Information Needed - Memory follows WP signal which is always High (Protected). Optionally the Card info may specify all devices as Never writeable.
Switch Controlled	S	Protect	High	No WP Information Needed - Memory follows WP signal.
		No prot	Low	
Always/Never	AN	None	Low	Card info must specify devices (addresses) which ignore the WP signal and are Never writeable. The remaining devices follow the WP signal and are therefore Always writeable.
Always/Switch	AS	Protect	High	Card info must specify devices (addresses) which ignore the WP signal and are Always writeable. The remaining devices follow the WP signal.
		No prot	Low	
Never/Switch	NS	Protect	High	Card info must specify devices (addresses) which ignore the WP signal and are Never writeable. The remaining devices follow the WP signal.
		No prot	Low	
Always/Never/Switch	ANS	Protect	High	Card info must specify both devices (addresses) which ignore the WP signal and are Always writeable as well as the devices which ignore the WP signal and are Never writeable. The remaining devices follow the WP signal
		No prot	Low	

4.9 Timing Functions

4.9.1 Common Memory Timing Specification

This section describes Common Memory Access Timing.

4.9.2 Common Memory Read Timing for all types of Memory

There are several types of Memory Cards — SRAM, OTPROM, etc.— and within a memory card, several types of memory devices may be mounted. To maintain compatibility among several types of memory devices, read-timing specifications are common. The read-timing specifications are shown in Table 4-11.

Table 4-11: Common Memory Read Timing Specification for all types of Memory

Speed Version			600ns[1,2]		250ns[3]		200ns		150ns		100ns	
Item	Symbol	IEEE Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Read Cycle Time	tcR	t AVAV	600[2]		250[5]		200		150		100	
Address Access Time[4]	ta (A)	t AVQV		600[2]		250[3]		200		150		100
Card Enable Access Time	ta (CE)	t ELQV		600[2]		250[3]		200		150		100
Output Enable Access Time	ta (OE)	t GLQV		300[2]		125[3]		100		75		50
Output Disable Time from CE	tdis (CE)	t EHQX		150		100		90		75		50
Output Disable Time from OE	tdis (OE)	t GHQZ		150		100		90		75		50
Output Enable Time from CE	ten (CE)	t ELQNZ	5		5		5		5		5	
Output Enable Time from OE	ten (OE)	t GLQNZ	5		5		5		5		5	
Data Valid from Add Change[4]	tv (A)	t AXQX	0		0		0		0		0	
Address Setup Time[5]	tsu (A)	t AVGL	100		30		20		20		10	
Address Hold Time [5]	th (A)	t GHAX	35		20		20		20		15	
Card Enable Setup Time[5]	tsu (CE)	t ELGL	0		0		0		0		0	
Card Enable Hold Time[5]	th (CE)	t GHEH	35		20		20		20		15	
Wait Valid from OE[5]	tv (WT-OE)	t GLWTV		100		35		35		35		35
Wait Pulse Width [6]	tw (WT)	t WTLWTH		12us		12us		12us		12us		12us
Data Setup for Wait Released [6]	tv (WT)	t QVWTH	0		0		0		0		0	

[1] 600 nsec cycle times apply for 3.3 volt reduced operating voltage.

[2] 3.3V timing for cycles >600 μ n sec are equal to value given + (cycle time-600). All other parameters are identical.

[3] 5V timing for cycles >250μ n sec are equal to value given + (cycle time-250). All other parameters are identical.

[4] The –REG signal timing is identical to address signal timing.

[5] These timing are specified for hosts and cards which support the –WAIT signal.

[6] These timings specified only when –WAIT is asserted within the cycle.

[7] All timings measured at card. Skews & delays from the system driver/receiver to the card must be accounted for by the system.

4.9.3 Write Timing for SRAM Card

Write Timing Specs are shown in Table 4-12.

Table 4-12: Common Memory Write Timing Specification SRAM

Speed Version			600ns[3,6]		250ns[5]		200ns		150ns		100ns	
Item	Symbol	IEEE Symbol	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Write Cycle Time	tcW	t AVAV	600[6]		250[5]		200		150		100	
Write Pulse Width	tw (WE)	t WLWH	300[6]		150[5]		120		60		60	
Address Setup Time[4]	tsu (A)	t AVWL	50		30		20		20		10	
Address Setup Time for WE[4]	tsu (A-WEH)	t AVWH	350[6]		180[5]		140		100		70	
Card Enable Setup Time for WE	tsu (CE-WEH)	t ELWH	300[6]		180[5]		140		100		70	
Data Setup Time for WE	tsu (D-WEH)	t DVWH	150[6]		80[5]		60		50		40	
Data Hold Time	th (D)	t WMDX	70		30		30		20		15	
Write Recover Time	trec (WE)	t WMAX	70		30		30		20		15	
Output Disable Time from WE	tdis (WE)	t WLQZ		150		100		90		75		50
Output Disable Time from OE	tdis (OE)	t GHQZ		150		100		90		75		50
Output Enable Time from WE	ten (WE)	t WHQNZ	5		5		5		5		5	
Output Enable Time from OE	ten (OE)	t GLQNZ	5		5		5		5		5	
Output Enable Setup from WE	tsu (OE-WE)	t GHWL	35		10		10		10		10	
Output Enable Hold from WE	th (OE-WE)	t WHGL	35		10		10		10		10	
Card Enable Setup Time[1]	tsu (CE)	t ELWL	0		0		0		0		0	
Card Enable Hold Time[1]	th (CE)	t GHEH	35		20		20		20		15	
Wait Valid from WE [1]	tv (WT-WE)	t WLWTV		100		35		35		35		35
Wait Pulse Width [2]	tw (WT)	t WTLWTH		12us		12us		12us		12us		12us
WE High from Wait Released [2]	tv (WT)	t WTHWH	0		0		0		0		0	

Notes:
1) These timing are specified for hosts and cards which support the –WAIT signal.
2) These timings specified only when –WAIT is asserted within the cycle.
3) 600 nsec cycle times apply for 3.3 volt reduced operating voltage.
4) The –REG signal timing is identical to address signal timing.
5) 5V timing for cycles >250 µ *n* sec are equal to value given + (cycle time-250). All other parameters are identical.
6) 3.3V timing for cycles >600 µ *n* sec are equal to value given + (cycle time-600). All other parameters are identical.
7) All timings measured at card. Skews & delays from the system driver/receiver to the card must be accounted for by the system.

4.9.4 Common Memory Write Timing for OTPROM, EPROM, and Flash Memory

The programming specification of various memory devices are not standardized. Moreover, programming specifications may vary among different generations of the same device. Consequently, it is not practical to set standardized programming specifications for these memory cards.

4.9.5 Attribute Memory Read Timing Specification

The Attribute Memory's access time is defined as 300ns at standard operating voltage of Vcc=5V ±5%. Detailed timing specifications are shown in Table 4-13.

Table 4-13: Attribute Memory Read Timing Specification for all types of Memory

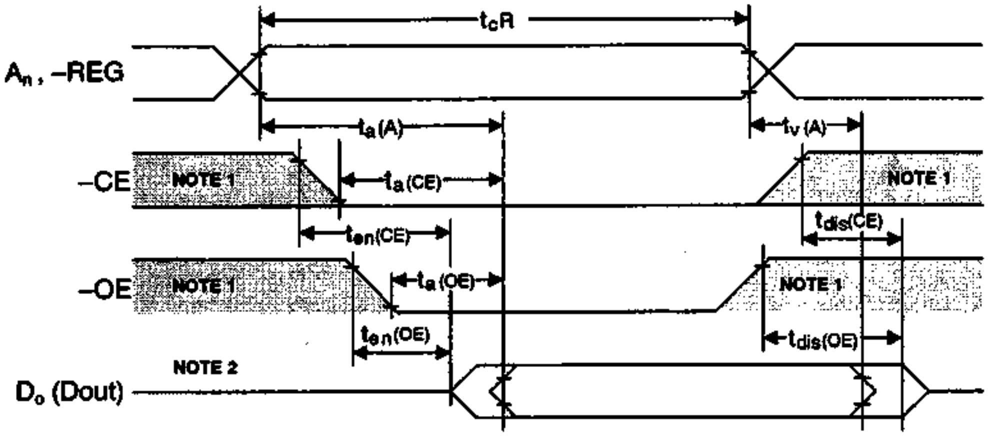
Speed Version			300ns	
Item	**Symbol**	**IEEE Symbol**	**Min**	**Max**
Read Cycle Time	tcR	t AVAV	300	
Address Access Time	ta (A)	t AVQV		300
Card Enable Access Time	ta (CE)	t ELQV		300
Output Enable Access Time	ta (OE)	t GLQV		150
Output Disable Time from CE	tdis (CE)	t EHQZ		100
Output Disable Time from OE	tdis (OE)	t GHQZ		100
Output Enable Time from CE	ten (CE)	t ELQNZ	5	
Output Enable Time from OE	ten (OE)	t GLQNZ	5	
Data Valid from Add Change	tv (A)	t AXQX	0	
Address Setup Time[1]	tsu (A)	t AVGL	30	
Address Hold Time [1]	th (A)	t GHAX	20	
Card Enable Setup Time[1]	tsu (CE)	t ELGL	0	
Card Enable Hold Time[1]	th (CE)	t GHEH	20	
Wait Valid from OE [1]	tv (WT-OE)	t GLWTV		35
Wait Pulse Width [2]	tw (WT)	t WTLWTH		12us
Data Setup for Wait Released [2]	tv (WT)	t QVWTH	0	

Notes: 1) These timing are specified for hosts and cards which support the –WAIT signal.
2) These timings specified only when –WAIT is asserted within the cycle.

4.9.6 Attribute Memory Write Timing Specification

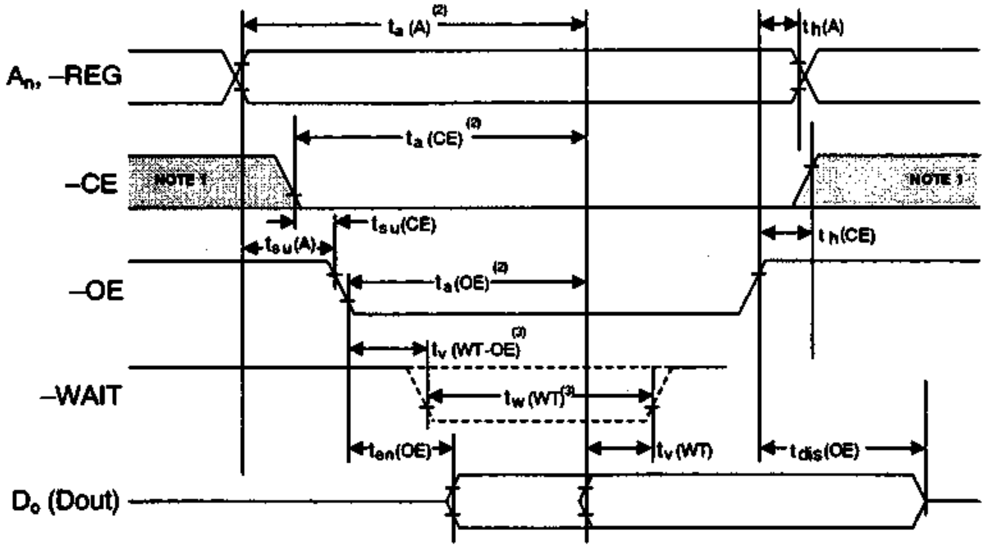
In the absence of other information, Attribute Memory Write timing will be 250 nsec SRAM timing.

4.9.7 Memory Timing Diagrams



Note 1: The shaded portion may be either high or low.
Note 2: Output Load = 1 TTL + 100pf.
Note 3: -WE remains high throughout this diagram.

Figure 4-1. Read Timing Diagram

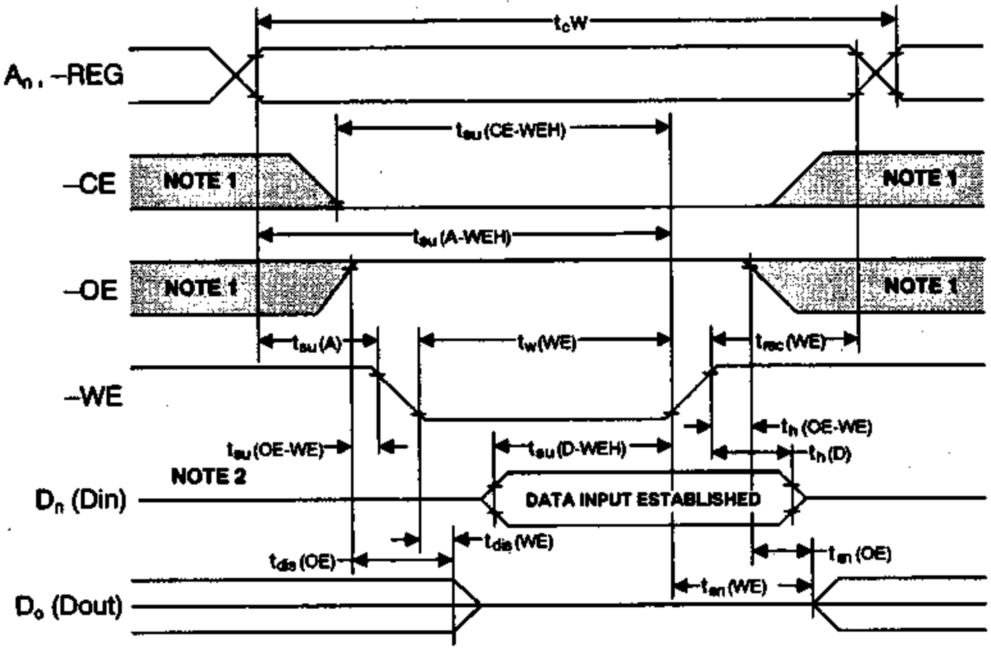


Note 1: The shaded portion may be either high or low.

Note 2: Applies to card only when -WAIT is not asserted by card. However, the host shall always provide at least this access time before sampling data.

Note 3: Applies only when -WAIT is asserted by card.

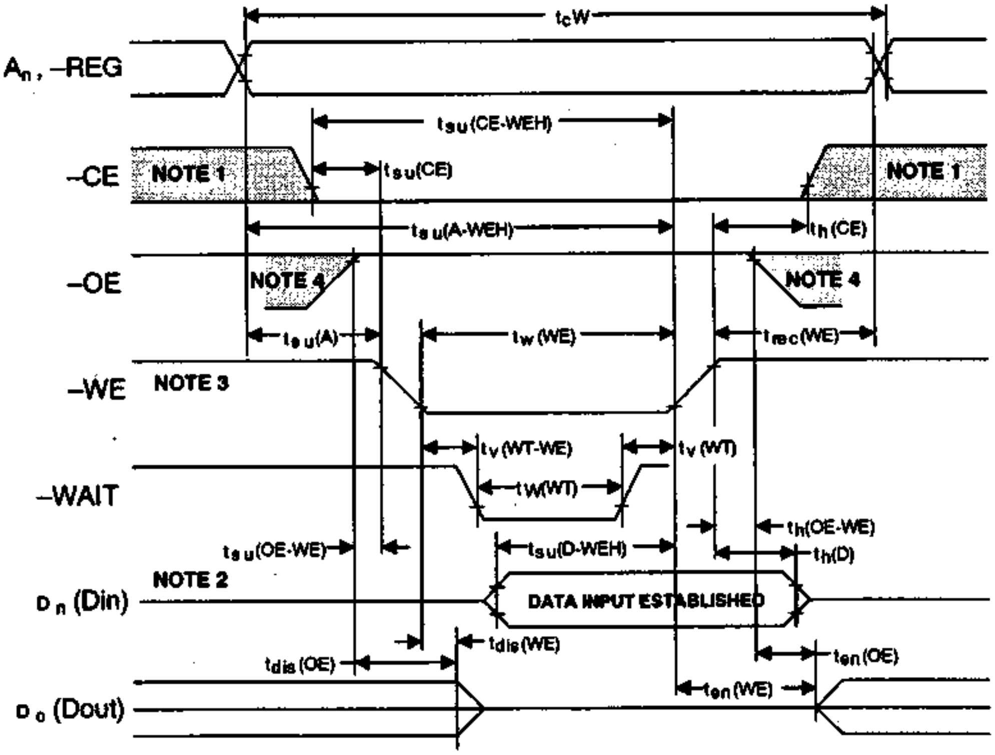
Figure 4-2. Read Timing Diagram (Wait Supported by Socket)



Note 1: The shaded portion may be either high or low.

Note 2: When the data I/O pin is in the output state, no signals shall be applied to the data pins (D0- D15) by the system.

Figure 4-3. Write Timing Diagram (WE control)



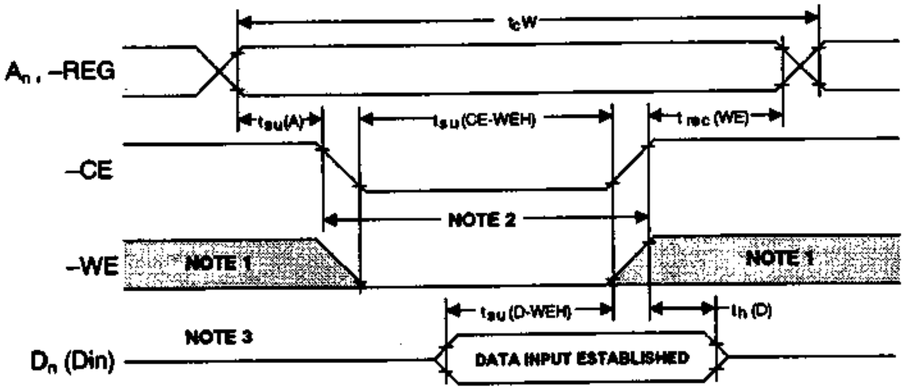
Note 1: The hatched portion may be either high or low.

Note 2: When the data I/O pin is in the output state, no signals shall be applied to the data pins (D0- D15) by the system.

Note 3: Minimum write pulse width must always be met whether or not card asserts -WAIT.

Note 4: May be high or low for Write timing, but restrictions on -OE timing in Figure 4-2 apply.

Figure 4-4. Write Timing Diagram (Wait Supported by Socket)



Note 1: The shaded portion may be either high ("H") or low ("L").

Note 2: OE must be high ("H").

Note 3: When the data I/O pin is in the output state, no signals shall be applied to the data pins (D0-D15) by the system.

Figure 4-5. Write Timing Diagram (CE control)

4.10 Electrical Interface

4.10.1 Signal Interface

Electrical specifications must be maintained to insure data reliability.

The Memory Card normal Operating Voltage for Vcc is 4.75 to 5.25 DC. Dual Operating Voltage Cards also permit a Reduced Operating Voltage for Vcc of 3.135 to 3.465. Interface signal levels are compatible with standard TTL or CMOS for the Operating Voltage used.

Table 4-14: Electrical Interface

Item	Signal	Card	Host	Card Output Format
Control Signal	-CE1 -CE2 -REG -IORD -IOWR	pull-up to Vcc R > 10K ohms and must be sufficient to keep inputs inactive when the pins are not connected at the host.[3]		
	-OE -WE/PGM	pull-up to Vcc R > 10K ohms*[3]		
	+RESET	pull-up to Vcc R > 100K ohms*		
	RFSH	NC	NC	Not Defined
Status Signal	+RDY/-BSY -INPACK -WAIT		pull-up[4]	
	+WP			
Address	A0-A25	pull-down R > 100K ohms*[5]		
Data Bus	D0-D15	pull-down R > 100K ohms*[2]	2	
Card Detect	-CD1 -CD2	connected to GND in the card	pull-up > 10K ohms	
Reserved Pin	RFU	NC	NC	
Battery/Detect	BVD1 BVD2		pull-up, NOTE 1	asserted or deasserted

*Resistor is optional

1. BVD2 was not defined in the JEIDA Version 3 document. Systems fully supporting JEDIA version 3 SRAM cards must pull up pin 62 (BVD2) to avoid sensing their batteries as "Low".
2. Data Signals: The host and each card shall present a load no larger than 50 pF at a DC current 450 µA low state and 150 µA high state. The host and each card shall be able to drive at least the following load while meeting all AC timing requirements: 100 pF with DC current 1.6mA low state and 300µA high state. This permits the host to wire two sockets in parallel without derating the card access speeds.
3. Control Signals: Each card shall present a load to the socket no larger than 50 pF at a DC current of 700 µA low state and 150 µA high state, including pullup resistor. The socket shall be able to drive at least the following load while meeting all AC timing requirements: (the number of sockets wired in parallel) multiplied by (50 pF with DC current 700 µA low state and 150 µA high state per socket).
4. Status Signals: The socket shall present a load to the card no larger than 50 pF at a DC current of 400 µA low state and 100 µA high state, including pullup resistor. The card shall be able to drive at least the following load while meeting all AC timing requirements: 50 pF at a DC current of 400 µA low state and 100 µA high state.
5. Address Signals: Each card shall present a load of no more than 100 pF at a DC current of 450A µA low state and 150 µA high state. The host shall be able to drive at least the following load while meeting all AC timing requirements: (the number of sockets wired in parallel) multiplied by (*100* ~~50~~ pF with DC current 450 µA low state and 150 µA high state per socket).

4.10.2 Memory Address Decoding

The PC Card's maximum memory-address space is 64 megabytes. The address bus is defined in A0-A25 with A0 being the LSB and A25 the MSB. Address bit A0 is a "don't care" when card is in word-access mode.

In the case of SRAM without Attribute Memory, address decoding is required as follows:

1. Minimum memory unit is 16KB
2. Memory address starts from 00H
3. Memory units exist continuously.

4.10.2.1 Card Configuration Registers Address Decoding

PC Cards that can be configured by the system — including cards which support I/O and those which exceed nominal system requirements — contain Card Configuration Registers located in the Attribute Memory space. The Card Configuration Registers are a set of numbered, standardized, 8-bit registers. These registers are addressed at consecutive even-byte addresses starting at the Base Address indicated in the Configuration Tuple. Whether Card Configuration Register number N is present on the card or not, the location of the register is always given by: Base Address + 2 * N. Refer to Section 4.15.

4.10.3 I/O Address Space Decoding

During I/O operations, the I/O address spaces of each PC Card may be either independent or overlapping. An independent I/O-address window is a portion of the system's I/O space which is assigned solely to a single PC Card and not shared with any other system resource. Each system must be able to allocate to each card an independent I/O-address window of at least 8 bytes of contiguous address space. The window must begin exactly on an 8-byte boundary. An overlapping I/O-address window is a portion of the system's I/O space which is assigned to any combination of PC Cards and other system resources. When the Overlapping Window method is used by the card, the card responds to only a limited number of the I/O addresses in the window. Also with this method, the card may permit the system to choose from a selection of addresses to which the card can respond.

During a card reset, and immediately following a card reset, a card will not respond to any I/O-read or I/O-write cycles until it has been configured.

The standard permits PC Cards which implement I/O functions to use either an Independent or Overlapping Window, or to permit selecting between the two methods. The types of I/O address decoding performed by the card, as well as the specific range of addresses to which the card will respond (in Overlapping I/O Window mode), are described in the Card Configuration Table located within the Card Information Structure. To select a particular card configuration, the system writes the index number of one table entry back to the Card Configuration Registers.

Refer to Sections 4.15 and 5.2.8 regarding card configuration.

In order to be able to accommodate cards which require higher number of I/O ports, such as LAN, VGA, multifunction cards and game cards, PCMCIA recommends that the system reserve 128 contiguous bytes of its I/O space.

Typically this will be available in systems that decode I/O addresses above 400H

PCMCIA recommends that when the system reserves addresses above 400H the base address will begin at 128 byte boundaries.

4.10.3.1 Independent I/O Address Window

Independent I/O-Address windows for PC Cards provide a straightforward mechanism for I/O-space allocation among PC Cards and other system resources installed in the system. The method can be applied to systems which permit I/O drivers to determine the card's I/O-port assignment at execution time.

For each PC Card socket, the system must reserve a minimum of 8 contiguous bytes of its I/O address space starting at a Base Address with A0 through A2 all zero.

[Note: This should not be interpreted as preventing a system from actually allocating a smaller or discontiguous space to a particular I/O card whose I/O-space requirements are met thereby.]

The location of the Independent I/O-Address Window for each PC Card socket may be fixed permanently, or may be allocated when an I/O card is detected in the socket. The system must include a software method permitting driver software (or application software, if I/O is permitted from applications) to determine a card's I/O space Base Address.

Cards which use the Independent I/O-Address Window need decode only enough address lines to distinguish among the I/O ports actually implemented on the card. When two identical cards are installed in the system, they automatically reside at different addresses. They rely upon the system to provide sufficient address decoding to prevent conflicts with other installed devices, and may be located anywhere in the I/O space of the system.

Example: An example of this concept is used in the EISA bus. Each EISA card slot is uniquely assigned the first 256 bytes of a corresponding 1 kilobyte of I/O space.

4.10.3.2 Overlapping I/O Address Window

The Overlapping I/O-Address Window is provided for those systems whose I/O drivers have their I/O addresses bound before execution time. This includes both systems for which execution-time binding is not possible, and those having existing drivers which do not take advantage of execution-time binding of I/O addresses.

The system assigns the same block(s) of its I/O space to each PC Card using the Overlapping I/O-Address Window. Other resources in the system may also use portions of the block. Each card and system resource is responsible for decoding addresses within the block and responding to I/O only within a unique subset of those addresses. This mechanism relies on the cards and system resources having non-overlapping subsets of addresses to which they respond.

When each PC Card is installed in a system that uses an Overlapping I/O-Address Window, the addresses used by the card must be compared with the addresses used by other installed PC Cards, and the addresses used by other system resources. If the addresses conflict, the new card may be gracefully rejected by the system.

A PC Card may allow operation at several alternative sets of addresses. The host selects the desired alternative from the Card Configuration Table and informs the card by writing the index of the selected entry to the Card Configuration Registers. Refer to Section 4.15.

Example: A desktop computer example of Overlapping I/O-Address Windows occurs in both ISA and EISA bus-based computers. In these systems all expansion slots are selected during I/O accesses to the last 768 bytes of each 1 kilobyte of I/O space. The expansion cards individually determine to which addresses within that range they will respond. The specific set of addresses is often set by configuration switches, or jumpers, on the expansion card. A service technician is responsible for ensuring, during installation, that no address conflicts will occur.

4.11 Card Detect

The Memory Card provides a means of allowing the system to detect card insertion and removal. Signal lines CD1 and CD2 are connected to GND in the card. A pull-up resistor must be connected to CD1 and CD2 on the system side.`

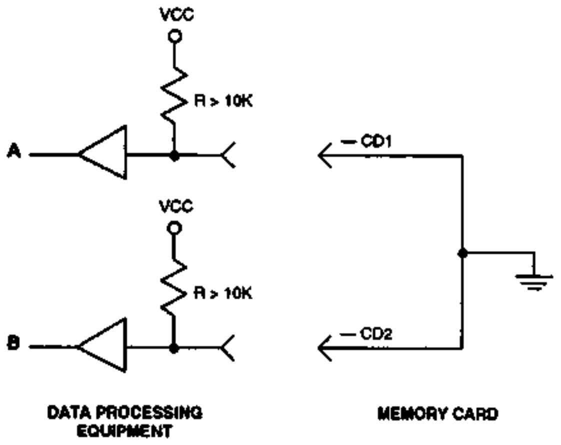


Figure 4-6. Card Detect

4.12 Battery Voltage Detect

When using SRAM Cards, it is critical for the system's data integrity to be able to determine the status of the on-card battery. The SRAM card provides two status signals for this purpose: BVD1 and BVD2. The Memory Card contains one or two voltage comparators and one or two reference voltages. The Memory Card compares the battery voltage with the reference voltages. Battery status is expressed on 2 digital signal lines, BVD1 and BVD2. If signal BVD2 is not supported, it is held to Vcc through a pull-up resistor on the card.

Table 4-15: Battery Voltage Detect

BVD1 (#63)	BVD2 (#62)	COMMENT
H	H	'GREEN' Battery Operational
H	L	'YELLOW' Battery should be replaced. Data is OK.
L	H	'RED' Battery & Data integrity is not guaranteed.
L	L	'RED' Battery & Data integrity is not guaranteed.*

* If BVD2 is not supported, BVD2 is held to Vcc and only one reference voltage is required.

4.13 Power-up And Power-down

4.13.1 Power-up/Power-down Timing

To retain data in the SRAM Card during power-up or power-down cycles, and to permit peripheral cards to perform power-up initialization, a timing specification is defined as follows.

Table 4-16: Power-up/Power-down Timing

			Value		
Item	Symbol	Condition	Min	Max	Unit
CE signal level[1]	Vi (CE)	0V < Vcc < 2.0V	0	ViMAX	V
		2.0V < Vcc <VIH	Vcc-0.1	ViMAX	
		VIH < Vcc	VIH	ViMAX	
CE Setup Time	tsu (Vcc)		20		ms
CE Setup Time	tsu (RESET)		20		ms
CE Recover Time	trec (Vcc)		0.001		ms
Vcc Rising Time[2]	tpr	10%--->90% of (Vcc +5%)	0.1	300	ms
Vcc Falling Time[2]	tpf	90% of (Vcc -5%)--->10%	3.0	300	ms
RESET Width	tw (RESET)		10		µs
	th (Hi-z Reset)		1		ms
	ts (Hi-z Reset)		0		ms

[1] ViMAX means Absolute Maximum Voltage for Input in the period of 0V < Vcc < 2.0V, Vi (CE) is only 0V~ViMAX

[2] The tpr and tpf are defined as "linear waveform" in the period of 10% to 90% or vice-versa, Even if the waveform is not "linear waveform", its rising and falling time must be met this specification.

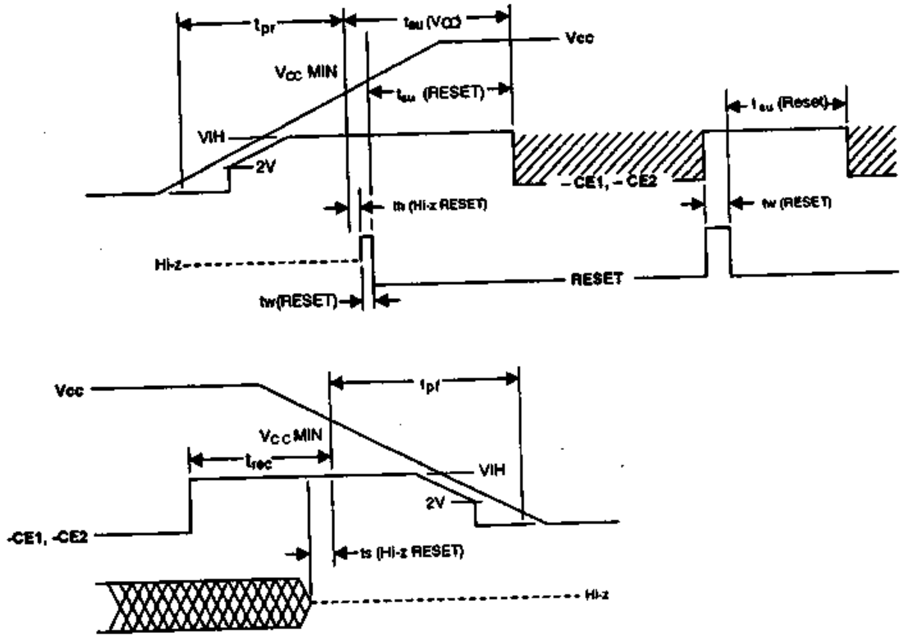


Figure 4-7. Power-Up/Down Timing for Systems Supporting RESET

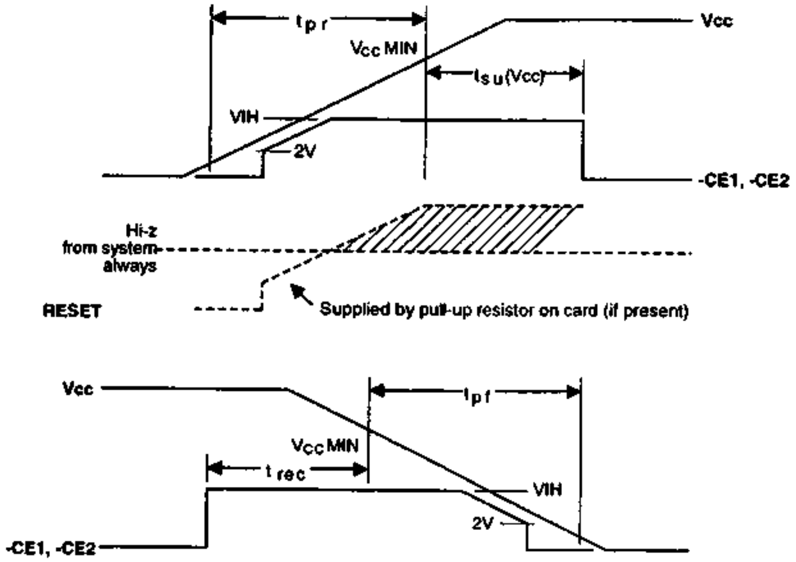


Figure 4-8. Power-Up/Down Timing for Systems Not Supporting RESET

4.13.2 Data Retention

This specification does not guarantee data retention in memory cards that conform to it. The conditions in the preceding tables indicate the minimum requirements to ensure data retention. Card vendors and system vendors may have to negotiate with each other to determine the detailed method of guaranteeing data retention for specific memory-card models.

4.13.3 Supplement

During card insertion or removal, with power active, its data-retention capability will depend on the individual Memory Card model, the manufacturer's environmental specifications, and other conditions. Therefore, there is no guarantee of data retention during card insertion or removal when under power.

4.14 I/O Function

This section describes the operation and configuration of I/O Cards inserted into a PCMCIA/JEIDA socket.

4.14.1 I/O Transfer Function

This section describes the operation of I/O transfer cycles (Input and Output Instructions) on I/O Cards.

4.14.2 I/O Input Function for I/O Cards

I/O-input transfers from I/O cards may be either 8-bit or 16-bit.

When a 16-bit transfer is attempted from a 16-bit port, the signal -IOIS16 must be asserted by the I/O Card. Otherwise, the -IOIS16 signal must be negated. When a 16-bit transfer is attempted, and the -IOIS16 signal is not asserted by the card, the system generates a pair of 8-bit references to access the word's even byte and odd byte.

An I/O card may extend the length of an input cycle by asserting the -WAIT signal at the start of the cycle.

Table 4-17: I/O Input Function for I/O Cards

Function Mode	-REG	-CE2	-CE1	A0	-IORD	-IOWR	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	High-Z	High-Z
Byte Input (8 bits)	L L	H H	L L	L H	L L	H H	High-Z High-Z	Even-Byte Odd-Byte
Word Access (16 bits)	L	L	L	L	L	H	Odd-Byte	Even-Byte
I/O Inhibit (e.g. during DMA)	H	X	X	X	L	H	High-Z	High-Z
High Byte Only	L	L	H	X	L	H	Odd-Byte	High-Z

Note: The Vpp1 and Vpp2 signals to the I/O Card are not required to be other than Vcc specifically for input transfers, however, they may be required to be at other voltage levels for proper card operation as indicated in the Card Information Structure. If the required Vpp levels cannot be provided by the system, the card may be rejected by the system.

4.14.3 I/O Output Function for I/O Cards

I/O-output transfers to I/O cards may be either 8-bit or 16-bit.

When a 16-bit transfer is attempted to a 16-bit port, the signal -IOIS16 must be asserted by the I/O Card. Otherwise, the -IOIS16 signal must be negated. When a 16-bit transfer is attempted, and the -IOIS16 signal is not asserted by the card, the system generates a pair of 8-bit references to access the word's even byte and odd byte.

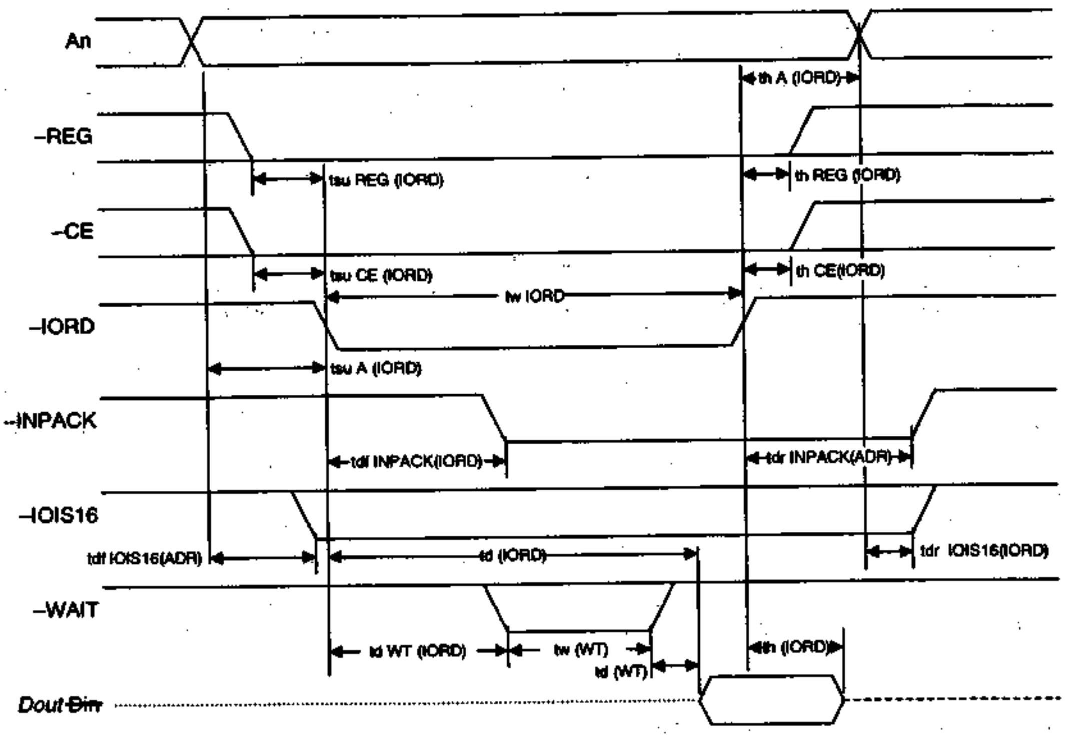
An I/O card may extend the length of an output cycle by asserting the -WAIT signal at the start of the cycle.

Table 4-18: I/O Output Function for I/O Cards

Function Mode	-REG	-CE2	-CE1	A0	-IORD	-IOWR	D15-D8	D7-D0
Standby Mode	X	H	H	X	X	X	X	X
Byte Input (8 bits)	L L	H H	L L	L H	H H	L L	X X	Even-Byte Odd-Byte
Word Access (16 bits)	L	L	L	L	H	L	Odd-Byte	Even-Byte
I/O Inhibit (e.g. during DMA)	H	X	X	X	H	L	X	X
High Byte Only	L	L	H	X	H	L	Odd-Byte	X

Note: The Vpp1 and Vpp2 signals to the I/O Card are not required to be other than Vcc specifically for output transfers, however, they may be required to be at other voltage levels for proper card operation as indicated in the Card Information Structure. If the required Vpp levels cannot be provided by the system, the card may be rejected by the system.

4.14.4 I/O Read (Input) Timing Specification



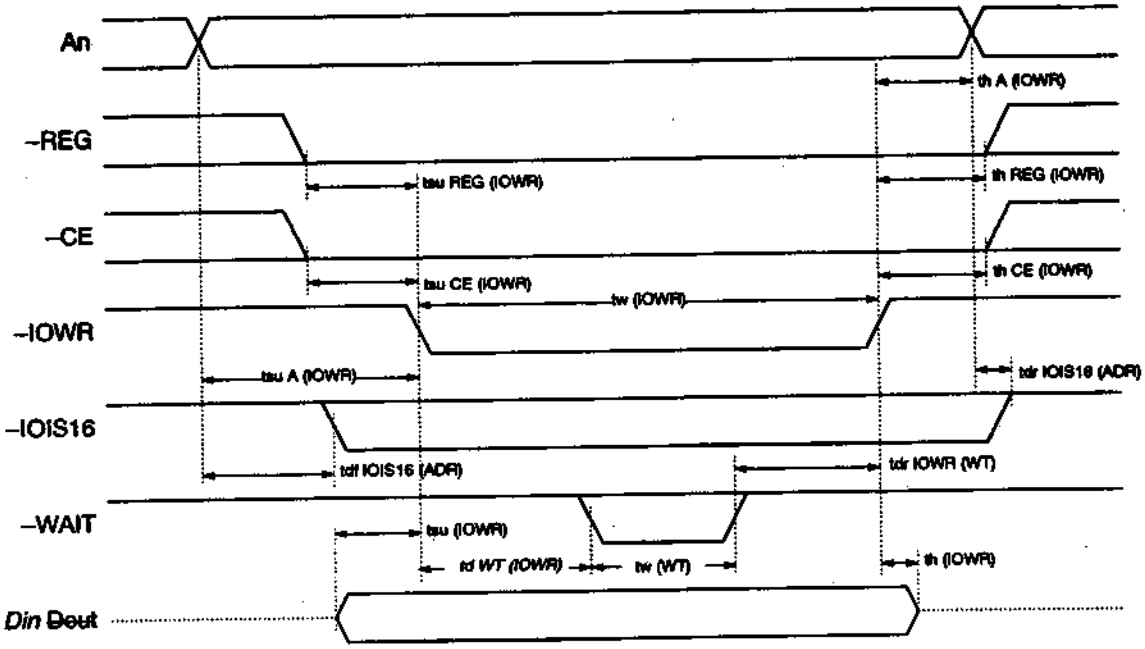
All timings are measured at the Card. Skews and delays from the system driver/receiver to the card must be accounted for by the system design.
Minimum time from -WAIT high to -IORD high is 0 nsec, but minimum -IORD width must still be met.
Dout signifies data provided by the card to the system.

Table 4-19: I/O Read (Input) Timing Specification for All 5V I/O Cards

Item	Symbol	IEEE Symbol	Min	Max
Data Delay after IORD	td (IORD)	t IGLQV		100
Data Hold following IORD	th (IORD)	t IGHQX	0	
IORD Width Time	tw IORD	t IGLIGH	165	
Address Setup before IORD	tsu A (IORD)	t AVIGL	70	
Address Hold following IORD	th A (IORD)	t IGHAX	20	
CE Setup before IORD	tsu CE (IORD)	t ELIGL	5	
CE Hold following IORD	th CE (IORD)	t IGHEH	20	
REG Setup before IORD	tsu REG (IORD)	t RGLIGL	5	
REG Hold following IORD	th REG (IORD)	t IGHRGH	0	
INPACK Delay Falling from IORD	tdf INPACK (IORD)	t IGLIAL	0	45
INPACK Delay Rising from IORD	tdr INPACK (IORD)	t IGHIAH		45
IOIS16 Delay Falling from Address	tdf IOIS16 (ADR)	t AVISL		35
IOIS16 Delay Rising from Address	tdr IOIS16 (ADR)	t AVISH		35
Wait Delay Falling from IORD	td WT (IORD)	t IGLWTL		35
Data Delay from Wait Rising	td (WT)	t WTHQV		35
Wait Width Time	tw (WT)	t WTLWTH		12,000

Note: The maximum load on WAIT, INPACK and IOIS16 are 1 LSTTL with 50 pF total load.
All timing in nsec.

4.14.5 I/O Write (Output) Timing Specification



All timings are measured at the Card.
Skews and delays from the system driver/receiver to the card must be accounted for by the system design.
Minimum time from -WAIT high to *-IOWR high* ~~-IORD~~ is 0 nsec, but minimum -IOWR timing must still be met.
Din signifies data provided by the system to the card.

Table 4-20: I/O Write (Output) Timing Specification for All 5V I/O Cards

Item	Symbol	IEEE	Min	Max
Data Setup before IOWR	tsu (IOWR)	t DVIWL	60	
Data Hold following IOWR	th (IOWR)	t IWHDX	30	
IOWR Width Time	tw IOWR	t IWLIWH	165	
Address Setup before IOWR	tsu A (IOWR)	t AVIWL	70	
Address Hold following IOWR	th A (IOWR)	t IWHAX	20	
CE Setup before IOWR	tsu CE (IOWR)	t ELIWL	5	
CE Hold following IOWR	th CE (IOWR)	t IWHEH	20	
REG Setup before IOWR	tsu REG (IOWR)	t RGLIWL	5	
REG Hold following IOWR	th REG (IOWR)	t IWHRGH	0	
IOIS16 Delay Falling from Address	tdf IOIS16 (ADR)	t AVISL		35
IOIS16 Delay Rising from Address	tdr IOIS16 (ADR)	t AVISH		35
Wait Delay Falling from IOWR	td WT (IOWR)	t IWLWTL		35
Wait Width Time	tw *(WT)*	t WTLWTH		12,000
IOWR high from Wait High	*tdr IOWR (WT)*	*t WTHIWH*	*0*	

Note: The maximum load on WAIT, INPACK and IOIS16 are 1 LSTTL with 50 pF total load.
All timing in nsec.

4.15 Card Configuration

Each configurable card is identified by a Card Configuration Table in the card's Card Information Structure. These cards must have one or more of a set of Card Configuration Registers which are used to control the configurable characteristics of the card. The configurable characteristics include the electrical interface, I/O-address space, interrupt request, and power requirements of the card.

All of the Card Configuration Registers may be both read and written. The registers are each one byte in size and located only on even-byte addresses to ensure their single-cycle access by both 8-bit and 16-bit systems.

These registers also provide a method for accessing some status information about a card. The information may be used to arbitrate between multiple-interrupt sources on the same interrupt request level. It may also be used to access status information which appears on pins 16, 33, 62 and 63 (Ready/Busy, Write Protect, and Battery Voltage Detect 1 and 2) in Memory Only cards

Table 4-21: Card Memory Spaces

-CE1	-REG	-OE	-WE	Address Offset	A0	Selected Register or Space
H	X	X	X	X	X	Standby
L	H	L	H	X	X	Common Memory Read
L	H	H	L	X	X	Common Memory Write
L	L	L	H	X	L	Card Info Structure or Configuration Register Read
L	L	H	L	X	L	Card Info Structure or Configuration Register Write
L	L	X	X	X	H	Invalid Access

Table 4-22: Card Configuration Registers

-CE1	-REG	-OE	-WE	Address Offset[1]	A0	Selected Register or Space	Reg #
L	L	L	H	NNNN0	L	Configuration Option Register Read	0
L	L	H	L	NNNN0	L	Configuration Option Register Write	0
L	L	L	H	NNNN2	L	Card Configuration and Status Register Read	1
L	L	H	L	NNNN2	L	Card Configuration and Status Register Write	1
L	L	L	H	NNNN4	L	Pin Replacement Register Read	2
L	L	H	L	NNNN4	L	Pin Replacement Register Write	2
L	L	L	H	NNNN6	L	Socket and Copy Register Read	3
L	L	H	L	NNNN6	L	Socket and Copy Register Write	3

Note 1:NNNN0 is the Configuration Registers Base Address specified in the TPCC_RADR field of the Configuration Tuple.

4.15.1 Configuration Option Register

The Configuration Option Register is used to configure the card and to issue a soft reset to the card. The register is a read/write register which contains two fields. The Configuration Option Register must be implemented in all configurable cards.

The Configuration Option Register is organized as follows:

Table 4-23: Configuration Option Register

D7	D6	D5	D4	D3	D2	D1	D0
SRESET	LevlREQ	Configuration Index					

The fields are as follows:

SRESET	SRESET Card. Setting this bit to one (1) places the card in the reset state. This is equivalent to assertion of the +RESET signal except that this bit is not cleared. Returning this bit to zero (0) leaves the card in the same unconfigured, reset state as following a power-up and hardware reset. This bit is set to zero by power up and hardware reset.
LevlREQ	Level Mode Interrupts selected when bit is one (1), Pulse Mode Interrupts selected when bit is zero (0).
Conf Index	Configuration Index. This field is written with the index number of the entry in the card's Configuration Table which corresponds to the configuration which the system chooses for the card. When the Configuration Index is 0, the card's I/O is disabled and will not respond to any I/O cycles, and will use the Memory Only Interface.

4.15.2 Card Configuration and Status Register

The Card Configuration and Status Register is an optional register which contains information about the card's condition.

Table 4-24: Card Configuration and Status Register Organization

D7	D6	D5	D4	D3	D2	D1	D0
Changed	SigChg	IOis8	Reserved (0)	Audio	PwrDwn	Intr	Reserved (0)

Changed	This bit indicates that one or more of the Pin Replacement Register bits CBVD1, CBVD2, CRdy/-Bsy or CWProt is set to one. When the Changed bit is set, the BVD1 pin is held low if the SigChg bit is 1 and the card is configured.
SigChg	This bit is set and reset by the host to enable and disable a state-change "signal" from the status register. When this bit is set and the card is configured for the I/O Interface, the Changed bit controls pin 63 and is called the Changed Status signal. If no state change signal is desired, this bit should be set to zero and the BVD1 (-STSCHG) signal will be held high while the card is configured for I/O.
IOis8	When the Host can provide I/O cycles only with an 8-bit D0-D7 data path, the Host shall set this to 1. The card is guaranteed that accesses to 16-bit registers will occur as two, byte accesses rather than a single 16-bit access. This information is useful when 16-bit and 8-bit registers overlap.
Reserved	Reserved bits must be 0
Audio	This bit is set to one to enable audio information on the BVD2 pin while the card is configured.
PwrDwn	This bit is set to one to request that the card enter a power-down state, if any. The system shall not place the card into a power-down state while the card's +RDY/-BSY line is in the low (Busy) state.
Intr	This bit represents the internal state of the interrupt request. This value is available whether or not interrupts have been configured. This signal remains true until the condition which caused the interrupt request has been serviced.

The Card Status Register must be implemented if the card generates audio, shares interrupts or requires other status information available in the register.

4.15.3 Pin Replacement Register Organization

The pin replacement register is used to provide the card status information which is otherwise provided on pins 16, 33, 62 and 63 on the Memory-Only Interface.

The register may be read and written, however, when written the lower 4 bits act as mask bits for changing the corresponding bit of the upper 4 bits.

The upper 4 bits are set when the corresponding bit in the lower 4 bits changes state.

Table 4-25: Pin Replacement Register Organization

D7	D6	D5	D4	D3	D2	D1	D0
CBVD1	CBVD2	CRdy/-Bsy	CWProt	RBVD1	RBVD2	RRdy/-Bsy	RWProt

CBVD1, CBVD2	These bits are set to one when the corresponding bit RBVD1 and/or RBVD2 change state. These bits may also be written by the host.
CRdy/-Bsy	This bit is set to one when the bit RRdy/-Bsy changes state. This bit may also be written by the host.
CWProt	This bit is set to one when the bit RRWProt changes state. This bit may also be written by the host.
RBVD1, RBVD2	When read, these bits represent the internal state of the Battery Voltage Detect circuits on cards which contain a battery. They correspond to the values which would be on pins 63 and 62, BVD1 and BVD2 respectively. When this bit is written as 1 the corresponding "changed" bit is also written. When this bit is written as 0, the corresponding changed bit is unaffected.
RRdy/-Bsy	When read, this bit represents the internal state of the Ready/-Busy signal. This bit may be used to determine the state of Ready/-Busy as that pin has been reallocated for use as Interrupt Request on IO Cards. When this bit is written as 1 the corresponding "changed" bit is also written. When this bit is written as 0, the corresponding changed bit is unaffected.
RWProt	This bit represents the state of the Write Protect switch. This signal may be used to determine the state of the Write Protect switch when pin 24 is being used for -IOIS16. When this bit is written as 1 the corresponding "changed" bit is also written. When this bit is written as 0, the corresponding changed bit is unaffected.

The Pin Replacement Register must be implemented if the card requires information about +Ready/-Busy, Write Protect or the Battery Voltage Detect status while the Memory Only Interface is not active. A logic 1 permits writing the corresponding bits; a logic 0 inhibits writing the corresponding bits.

4.15.4 Socket and Copy Register

This is an optional read-write register which the card may use to distinguish between similar cards installed in a system. This register, if present, is always written by the system before writing the card's Configuration Index Register.

Table 4-26: Socket and Copy Register Organization

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (0)	Copy Number			Socket Number			

Reserved	This bit is reserved for future standardization. This bit must be set to zero (0) by software when the register is written.
Copy Number	Cards which indicate in their CIS that they support more than one copy of identically configured cards, should have a copy number (0 to MAX twin cards, MAX=n-1) written back to the socket and copy register. This field indicates to the card that it is "n'th" copy of the card installed in the system which is identically configured. The first card installed receives the value 0. This permits identical cards designed to do so to share a common set of I/O ports while remaining uniquely identifiable. and consecutively ordered.
Socket Number	This field indicates to the card that it is located in the n'th socket. The first socket is numbered 0. This permits any cards designed to do so to share a common set of I/O ports while remaining uniquely identifiable.

4.16 Future Tasks And Remarks

4.16.1 Insertion/Removal with Power Active

PCMCIA recognizes the market need for card insertion or removal while under power. PCMCIA has been discussing this issue but has not yet achieved consensus on a specification which would guarantee a card's data retention during active-power insertion or removal. PCMCIA will continue to discuss this issue at future meetings.

4.16.2 Standardization of EPROM and EEPROM Programming

Programming specifications for EPROM and EEPROM are not yet standardized at the device level. Instead, the programming voltage, timing, and other conditions vary with individual vendors. Consequently, it is impossible to standardize at the memory-card level. PCMCIA urges device vendors to standardize their programming voltage, timing, and other conditions. The memory-card committee will continue to work towards a standard memory card which is reliable and easy to use.

4.16.3 Wide Operating Voltage

There is a large potential market of memory cards for battery-powered equipment. Low-voltage operation is urgently needed by this market. To respond to these needs, PCMCIA has specified a Dual Operating Voltage card (5 Volt and 3.3 Volt) and is investigating additional means by which low-voltage memories can be supported by the standard.

4.16.4 I/O Functionality

PCMCIA has defined an I/O interface as described in this document. This I/O functionality allows a variety of I/O cards — such as communications cards or disk emulation cards — to be implemented. The PCMCIA is continuing to address issues of additional I/O functions.

SECTION - 5

CARD METAFORMAT

CARD METAFORMAT

5.1 The Metaformat[1]

5.1.1 Goals of This Standard

The following goals guided the development of this standard.

1. The standard must support several different file-system formats on a single card, both DOS compatible as well as other file systems (e.g., XENIX). It should also support applications such as data storage for VCRs or musical instruments, which might not use any traditional file system to record their data. At the same time, any computer system should be able to look someplace on a card and determine such things as the card's overall size, type and other low-level information.

   Given the wide potential scope of applications for memory cards, the ability to read non-DOS cards on DOS-based systems will be of significant value to users.

   The ability to detect that a given card is formatted (though perhaps not readable by the computer into which it is plugged) is particularly valuable in that it allows system designers to protect users against common mistakes. Such a capability could be used, for example, to inform the user during the format routine that the card is already formatted as a data-storage card for a VCR.

2. Because application requirements differ, the standard should support various low-level data recording strategies (akin to physical formatting for floppies). These strategies would include sequential recording of blocks of bytes with no error checking; sequential recording of blocks of bytes with embedded error checking (CRC codes); sequential recording of bytes with separate error checking (e.g., non-sequential checksum bytes); or sequential non-blocked recording of bytes.

3. For compatibility with existing operating systems and application programs, the standard should be most focused on those environments where all media are organized in a disk-like way (i.e. with sectors, tracks and cylinders). On the other hand, the standard should also support those environments that simply address media as sequences of blocks.

4. The standard should support cards that include directly-executable ROM images, and cards that include a mixture of directly-executable images and DOS file systems.

5. The standard should support cards for the DOS environment that include programs that can be directly executed from ROM, or executed from RAM in the usual fashion, depending on the capabilities of the computer system.

6. The standard should be reasonably general, and should allow for future expansion without major rewriting of existing software. At the same time, for common (MS-DOS) environments, the standard should not be excessively general.

5.1.2 Overview

PCMCIA PC Card Standard, Release 2.01 Metaformat goals include the ability to handle numerous, somewhat incompatible data-recording formats and data organizations. As is done with networking standards, the Metaformat is a hierarchy of layers. Each layer has a number, which increases as the level of abstraction gets higher.

[1] The glossary in Appendix A defines many of the technical terms in this chapter.

The layers are:

0. The Physical Layer — the lowest layer of possible standardization. This layer specifies the interface and electrical characteristics of PC Cards.

1. The Basic Compatibility Layer specifies a minimal level of card-data organization. To be compatible at this level, each card should contain a small Card Information Structure ("CIS") or conform to the requirements of Section 5.5.3. This structure contains certain Level-1 information, primarily some fundamental information about the card's devices, such as size, speed, and the like. The information contained in the Card Information Structure is commonly called the Metaformat. In addition, this structure contains information about the card's organization at Levels 2, 3, and 4. References to Levels 1 through 4 pertain to levels of compliance.

   A card can comply at Level 1 without being required to comply at any higher level. Thus, *PCMCIA PC Card Standard,* Release 2.01 is an open standard. Cards that comply only at Level 1 need not reserve space for the higher-level information.

   The CIS can be thought of as being separate from the data recorded on the media. Under DOS, only the BIOS (or device driver) would be aware of its existence.

   The information block must be located such that it can be easily found by low-level software. This standard requires that the primary CIS be recorded in Attribute Memory starting at address zero.

   The first tuple starting from address 0 in the ATTRIBUTE memory space must be either a "Device Information tuple" (tuple code 01H), a "Null Control tuple" (tuple code 00H), or an "End of List tuple" (tuple code FFH, see Section 5.2.5.5 for processing).

   For flexibility, the CIS can be extended into Common Memory. This allows application parameters to be changed by the user. To facilitate automatic identification of "blank" cards, Attribute Memory can be read-only memory.

   At this level, the standard defines two kinds of information:

   - Data structures and concepts used by all layers of this standard, and
   - Physical device information.

2. The Data Recording Format Layer specifies how the data on the card is organized at the lowest level. This layer is analogous to the physical format of a floppy disk.

   The use of a traditional DOS file system or boot block, is not specified (or required) for compatibility at Level 2.

   Specific formats supported are:

   - Blocked, Unchecked — the bytes are recorded in blocks with no error checking,
   - Blocked, Checksummed — the bytes are recorded in blocks with checksums for error checking,
   - Blocked, with CRC — the bytes are recorded in blocks with CRC codes for error checking,
   - Unblocked — individual bytes of the card may be accessed or modified by software directly at random. The bytes are recorded in a way that does not correspond to a disk organization.

3. The Data Organization Layer specifies how the data is logically organized on the card. Possibilities are:

   - DOS (or other operating system) file system,
   - Flash file system,
   - Execute-In-Place (XIP) ROM image [see Section 6],
   - Application-specific organization.

A DOS file system can be used with any of the appropriate (blocked) Level- 2 organizations.

4. The System-Specific Layer defines standards that by their nature are specific to a particular operating environment.
   - The DOS XIP Standard defines a standardized way of preparing DOS-executable images on ROM cards. Programs that conform to this standard will execute correctly on any system that can read the ROM card and includes the appropriate address translation hardware and support software. A standard "RAM execution".EXE or.COM file can also be loaded into system RAM memory from a ROM card if XIP is not supported. See Section 6.

5.1.3 Vendor-Specific Information

Vendor-specific information allows card and software vendors to implement proprietary functions while remaining within the general framework of this standard.

Vendor-specific information is of two kinds:

- Vendor-specific fields are areas reserved in the data structures for free use by vendors. These fields have no meaning to the standard software.
- Vendor-specific codes are encoding values reserved to represent non-standard values in standard fields. In the absence of other information, standard software must interpret vendor-specific codes as meaning "the information in this field is not specified."

The card-manufacturer field in the CIS gives (knowledgeable) system software enough information to interpret vendor-specific fields and code values in the card Physical-Description tuples.

Similarly, the OEM and INFO fields in the CISTPL_VERS_2 tuple give (knowledgeable) system software enough information to interpret vendor-specific fields and code values in the card Logical-Format tuples.

In general, a system will not be able to interpret all possible vendor-specific fields or code values.

5.1.4 System Rejection of Unsupported Cards

This standard requires the following behavior when a system encounters an unrecognized vendor-specific field:

- If the unrecognized field itself is vendor-specific, the system shall ignore that field.
- If a standard field contains an unrecognized vendor-specific code, the system must refuse to perform any operation that requires the information encoded in that field.

5.2 Basic Compatibility (Layer 1)

This layer is the cornerstone of the standard. Any card that complies with this standard shall have at least a rudimentary Card Information Structure (referred to as the "CIS") starting at address zero of the card's Attribute-Memory space.

The Card Information Structure is a variable-length chain (or linked list) of data blocks or *tuples*. All tuples have the format shown in Table 5-1. One or more chains of tuple lists can be used. Longlink tuples are used to connect chains (see Section 5.2.5.2).

Table 5-1: Tuple Format

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Tuple code: CISTPL_xxx; see Table 5-2.					
	TPL_LINK		Offset to next tuple in list. This can be viewed as the number of additional bytes in tuple, excluding this byte. (n-1)					
			Bytes specific to this tuple.					

Byte 0 of each tuple contains a tuple code. A tuple code of FFH is a special mark indicating that there are no more tuples in the list. Byte 1 of each tuple contains a link to the next tuple in the list. If the link field is zero, then the tuple body is empty. If the link field contains FFH, then this tuple is the last tuple in the list.

There are thus two ways of marking the end of the tuple list: A **tuple code** of FFH, or a **tuple link** of FFH. See Section 5.2.5.5 for end of tuple list processing.

An End of List tuple should be used if the tuple list is intended to be extendable. A link field of FFH conserves one (1) byte of memory space but does not describe the actual length of the last tuple in the chain. It is recommended that CIS chains be terminated with an End of List tuple.

The use of an FFH link value is allowed for backward compatibility, but it is recommended to use the end of list tuple whenever possible. System software must use the link field to validate tuples. No tuple can be longer than 256 bytes. Some tuples provide a termination or stop byte that marks the end of the tuple. In this case, the tuple can effectively be shorter than the value implied by its link field. However, software must not scan beyond the implied length of the tuple, even if a termination byte has not been seen.

The following tuple codes are defined:

Table 5-2: Tuple Codes

Code	Name	Description
Layer 1 Basic Compatibility Tuples		
00H	CISTPL_NULL	Null tuple - Ignore
01H	CISTPL_DEVICE	The device information tuple (common memory)
02H-07H	-	(Reserved for future, upward-compatible versions of the device information tuple).
08H-0F	-	(Reserved for future, incompatible versions of the device information tuple).
10H	CISTPL_CHECKSUM	The checksum control tuple.
11H	CISTPL_LONGLINK_A	The long-link-control tuple (to Attribute Memory).
12H	CISTPL_LONGLINK_C	The long-link-control tuple (to Common Memory).
13H	CISTPL_LINKTARGET	The link-target-control tuple.
14H	CISTPL_NO_LINK	The no-link-control tuple.
15H	CISTPL_VERS_1	Level 1 version/product-information tuple.
16H	CISTPL_ALTSTR	The alternate-language-string tuple.
17H	CISTPL_DEVICE_A	Attribute Memory device information.
18H	CISTPL_JEDEC_C	JEDEC programming information (for Common Memory).
19H	CISTPL_JEDEC_A	JEDEC programming information (for Attribute Memory).
1AH	CISTPL_CONFIG	The Configuration tuple.
1BH	CISTPL_CFTABLE_ENTRY	The Configuration-Table-Entry tuple.
1CH	CISTPL_DEVICE_OC	Other operating conditions device information for Common Memory.
1DH	CISTPL_DEVICE_OA	Other operating conditions device information for Attribute Memory.
1EH	*CISTPL_DEVICE_GEO*	*Device geometry info tuple.*
1FH	*CISTPL_DEVICE_GEO_A*	*Device geometry infor tuple.*
Layer 2 Data Recording Format Tuples		
20H	*CISTPL_MANFID*	*Manufacturer Identification Tuple*
21H	*CISTPL_FUNCID*	*Function Identification tuple*
22H	*CISTPL_FUNCE*	*Function Extension Tuples*
23H	*CISTPL_SWIL*	*Software interleave tuple.*
24H-3FH	-	*(Reserved for future standardization)*
40H	CISTPL_VERS_2	The level-2 version tuple.
41H	CISTPL_FORMAT	The format tuple.
42H	CISTPL_GEOMETRY	The geometry tuple.
43H	CISTPL_BYTEORDER	The byte order tuple.
44H	CISTPL_DATE	The card initialization date tuple.
45H	CISTPL_BATTERY	The battery replacement date tuple.
Layer 3 Data Organization Tuples		
46H	CISTPL_ORG	The organization tuple.
47H-7FH	-	(Reserved for future standardization).
Layer 4 System-Specific Standard Tuples		
80H-FEH	-	Vendor unique tuples*
FFH	CISTPL_END	The end-of-list tuple.

* Note: The use of vendor-specific tuples is under review and subject to change. PCMCIA recommends restricting their use to the 80-8FH range.

PCMCIA expects that the CIS will be written once when the card is formatted and then rarely (if ever) updated. The standard is not designed to allow incremental updating of the CIS on Flash media. On EEPROM devices that require the CIS to be erased occasionally (for example when a Flash-type file system is reorganized), it is suggested that a buffer-page strategy be used, with an appropriate utility that can recover from a power-failure.

Note that most implementations will be limited to reading cards of a specific format, or at most, of a few different formats. Thus, many combinations of values available in the tuples will be non-portable. PCMCIA suggests that implementors restrict themselves to the suggested low-level formats presented in Section 5.3.3.

5.2.1 Byte Order Within Tuples

Within tuples, all multi-byte numeric data shall be recorded in little-endian order. That is, the least-significant byte of a data item shall be stored in the first byte of a given field.

Within tuples, all character data shall be stored in the natural order. That is, the first character of the field shall be stored in the first byte of the field. Fixed-length character fields shall be padded with null characters, if necessary.

5.2.2 Byte Order on Wide Cards

If a card has a data path wider than 8-bits, one must assign a byte order to the data path, at least for fields within the CIS that are recorded in Common Memory space.[1] This standard requires that the low-order byte of word 0 be used to record byte 0 of the CIS. Ascending bytes of each word shall be used to sequentially record bytes from the CIS. When the first word is filled, the same process shall be repeated on subsequent words until the entire CIS is recorded. On Intel-family machines, this byte order is equivalent to the native order; other machines may need to reorder the bytes when reading or writing the CIS.

The basic compatibility layer does not impose any particular byte order on non-header portions of the card. However, some data-format layers will impose further requirements.

5.2.3 Tuple Format in Attribute Memory Space

PC Cards have two address spaces: *Attribute-Memory space* and *Common-Memory space*. The electrical specification for PC Cards requires that information be placed only in even-byte addresses of Attribute-Memory space. The contents of odd-byte addresses of Attribute-Memory space are not defined.

For simplicity, this specification describes the tuples of the Metaformat as if the bytes of each tuple were recorded consecutively. When a tuple is recorded in Common-Memory space, the bytes will indeed be recorded consecutively. However, when a tuple is recorded in Attribute-Memory space, the data will be recorded in even bytes only.

Link fields of tuples stored in Attribute-Memory space are handled as follows. If only the even bytes are read, and the tuples are packed into consecutive bytes in system memory, the link fields shall be set appropriately for byte addressing. This means that the link-field values are conceptually the same whether a tuple resides in Common or in Attribute memory. However, this does mean that if Attribute Memory is directly addressed, the offset to the next tuple in Attribute Memory is [2*link field].

5.2.4 Use of Common Memory Space for Attribute Memory Storage

For cost reasons, many ROM cards and some cards of other types will not implement a separate Attribute-Memory space. On these cards, regardless of the state of the -REG line, memory cycles will always access Common Memory. These cards will provide an Attribute-Memory-style CIS starting at byte zero of the card, and recorded in even bytes only. If, for space reasons, the manufacturer wants to switch to a Common-Memory-style CIS (packed into ascending bytes), a

[1] At present, Attribute Memory is byte-wide only; only the even bytes are present.

long link to Common Memory shall be embedded in the CIS. The target address of this long link will be non-zero, and the Common-Memory CIS will be stored immediately following the Attribute-Memory CIS.

It is important to distinguish between Attribute-Memory *space* and Attribute Memory. All PC Cards will have Attribute-Memory space, accessed by asserting the -REG pin. In addition, some PC Cards will have a distinct physical Attribute Memory. In this case, the contents of location 0 in Attribute-Memory space will be different and distinct from the contents of location 0 in Common-Memory space. However, some PC Cards will not have Attribute Memory distinct from Common Memory. Here, memory-read operations from a given location in Attribute-Memory space will return the same data as read operations from the same location in Common-Memory space. Data accessed from Attribute-Memory space must be stored in the even bytes only, even if Attribute Memory is not distinct from Common Memory. Regardless of the presence or absence of Attribute Memory, the CIS for PC Cards always begins at location 0 of Attribute-Memory space.

This standard allows attribute information to be stored both in Attribute-Memory and Common-Memory space. Tuples stored in Common-Memory space are recorded in sequential bytes. Both the card's even and the odd bytes are used to record data.

Note: The use of odd bytes to represent tuple data is controlled by the logical-address space in which the tuple resides, not by the type of memory actually used to record the tuple. If the tuple is intended to be accessed via Attribute-Memory space, it must be stored only in the even bytes. If it is intended to be accessed via Common-Memory space, it must be stored in both even and odd bytes following a long-link target.

5.2.5 Control Tuples

Multiple instances of a tuple are allowed unless otherwise specified.

5.2.5.1 CISTPL_NULL: The Null Control Tuple

The null-control tuple is simply a placeholder. It has a non-standard form and consists solely of the code byte. See Table 5-3.

Table 5-3: The Null Control Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_NULL (00H): ignore this tuple.					

Software shall ignore these tuples. The next tuple begins at the next byte in sequence.

5.2.5.2 CISTPL_LONGLINK_A, CISTPL_LONGLINK_C: The Long-Link Control Tuples

The long-link tuples are used to jump beyond the limits of the 1-byte link field, from one tuple chain to another. The target tuple chain may be in Attribute-Memory or Common- Memory space, as indicated by the tuple code. See Table 5-4.

Table 5-4: Long Link Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Long-link tuple code (CISTPL_LONGLINK_A, 11H; or CISTPL_LONGLINK_C, 12H)					
1	TPL_LINK		Link to next tuple (at least 4).					
2 .. 5	TPLL_ADDR		Target address; stored as an unsigned long, low-order byte first.					

The tuple-code byte selects the new address space. For example, CISTPL_LONGLINK_A indicates that the target is in Attribute-Memory space; CISTPL_LONGLINK_C indicates Common-Memory space.

A given tuple list shall contain at most one long-link tuple. The long-link tuple need not appear as the last tuple in a given list because the entire list containing the long-link tuple will be processed before the link is honored.

Software shall verify that the long-link tuple points to a link-target tuple before processing the target list. Because a long-link tuple may point to uninitialized RAM, it is important that software simply reject target-tuple lists that do not begin with a link-target tuple.

5.2.5.3 CISTPL_LINKTARGET: The Link-Target Control Tuple

The link-target tuple is used for robustness. Every long-link tuple must point to a valid link-target tuple. The link-target tuple has one field— the string "CIS". The link field of the link target should always point to the next byte after the link-target tuple. Processing software is required to check that the link-target tuple is correct before deciding to process the tuple list at the new target address. See Table 5-5.

Table 5-5: Link Target Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_LINKTARGET (13_H)					
1	TPL_LINK		Link to next tuple (at least 3).					
2	TPLTG_TAG		"C" (43_H)					
3			"I" (49_H)					
4			"S" (53_H)					

5.2.5.4 CISTPL_NO_LINK: The No-Link Control Tuple

To save Attribute-Memory space, processing software shall assume the presence of a CISTPL_LONGLINK_C, 0L tuple as part of the primary tuple chain. This is the tuple chain which starts at address 0 of Attribute-Memory space. This assumption can be overridden by placing an explicit long-link tuple in the Attribute-Memory CIS. To prevent software from trying to execute any long-link operations, the card manufacturer can place a no-link tuple in the Attribute-Memory CIS. See Table 5-6.

Table 5-6: No-Link Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_NO_LINK (14_H)					
1	TPL_LINK		Link to next tuple (may be zero).					

[Note: The body of this tuple is always empty.]

A given tuple chain shall contain at most one no-link control tuple. No-link tuples and long-link tuples are mutually exclusive. A given chain may contain either a no-link tuple, or a long-link tuple, but not both.

5.2.5.5 CISTPL_END: The End-Of-List Tuple

The end-of-list control tuple marks the end of a tuple chain. It has a non-standard form and consists solely of the code byte. See Table 5-7.

Table 5-7: The End-of-List Control Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_END (FF_H): end of this tuple chain.					

Upon encountering this tuple, system software shall take one of the following actions.

- If a long-link tuple was encountered previously in this chain, continue tuple processing at the location specified in the long-link tuple.
- If processing a tuple chain (other than the primary CIS tuple chain), and no long-link tuple was seen in this chain, then no tuples remain to be processed.
- If processing the primary CIS tuple chain (the list starting at address 0 in Attribute- Memory space), and neither a long-link nor a no-link tuple were seen in this chain, then continue tuple processing as if a long-link to address 0 of Common-Memory space were encountered. For validation of the implied long link to COMMON memory, as with an explicit Long Link tuple, the secondary tuple chain in COMMON memory must begin with a valid Link Target tuple.
- If a no-link tuple was encountered previously in this chain, no tuples remain to be processed.

5.2.5.6 CISTPL_CHECKSUM: The Checksum Control Tuple

For additional reliability, the CIS can contain one or more checksum tuples. This tuple has three fields:

- The relative address of the block of CIS memory to be checked;
- The length of the block of CIS memory to be checked; and
- The expected checksum.

The checksum algorithm is a straight modulo-256 sum. Relative addressing is used to make the CIS, as a whole, position-independent. The checksum tuple can only validate memory in its own address space. See Table 5-8.

Table 5-8: Checksum Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_CHECKSUM (10H)					
1	TPL_LINK		Link to next tuple (at least 5).					
2 .. 3	TPLCKS_ADDR		Offset to region to be checksummed, stored with LSB first.					
4 .. 5	TPLCKS_LEN		Length of region to be checksummed, given with LSB first.					
6	TPLCKS_CS		The checksum of the region.					

The checksum is calculated by summing the bytes of the selected region using modulo 256. The result must match the value stored in byte 6 of the checksum tuple.

TPLCKS_ADDR contains the offset, relative to the start address of this tuple, of the region to be checksummed. The address is a signed, 2-byte integer. Negative values indicate locations prior to the checksum tuple; positive values indicate locations after the checksum tuple. The exact interpretation depends on the address space containing the tuple.

TPLCKS_LEN contains the number of bytes to be checksummed. The number is expressed as an unsigned, 2-byte integer.

If the tuple appears in Common-Memory space, the checksum is calculated in the obvious way. The contents of TPLCKS_ADDR (as a signed integer) are added to the base address of the tuple, yielding the target address. Starting at the target address, the algebraic sum is calculated of all the bytes included in the range. Then, the low-order 8-bits of this sum are compared to the value stored in TPLCKS_CS. If identical, the region of tuple memory covered by the checksum passes the checksum test.

If the tuple appears in Attribute-Memory space, the checksum operation is a bit more complicated. Again, the data structures are recorded in such a way as to minimize the differences between Attribute space representation and Common Memory representation. To form the target address, 2 * *offset* is added to the base-byte target address of the tuple. Then, the algebraic sum is formed of the even bytes in the address range [i.e. *target, target* +2**length-1*], ignoring the odd bytes. The low-order 8-bits of this sum is then compared to the value stored in TPLCKS_CS.

5.2.5.7 CISTPL_ALTSTR: The Alternate-Language String Tuple

Several tuples contain character strings which are intended to be displayed to the user only under certain circumstances. Some international applications need the ability to store strings for a number of different languages. Rather than having various languages used in the tuples, this standard provides alternate-string tuples. Strings in the primary tuples are always recorded in ISO 646 IRV code using characters in the range 20h-7Eh. Multiple instances of this tuple are allowed; each associates with most recent (previous) "NON-ALT STRING" tuple. Tuple codes affected are 15H (CISTPL_VERS_1) and 40h (CISTPL_VERS_2).

Alternate-string tuples contain two kinds of information:

- A code representing the language (an ISO-standard escape sequence), and
- A series of strings.

These strings are to be substituted for the primary strings when operating in a different language environment. See Table 5-9.

Table 5-9: Alternate Language String Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_ALTSTR (16H)					
1	TPL_LINK		Link to next tuple (at least p-1).					
2 .. m-1	TPLALTSTR_ESC		ISO-standard escape sequence to select the character set for these strings. Indicates which character set is associated with these strings. The leading ESCAPE is not recorded. Terminated by a NUL (00H). A special escape sequence denotes the PC Extended-ASCII character set; see appendix					
m .. n-1			Alternate string 1; translation for first string in most recent non-ALTSTR tuple. Terminated by 00H.					
n .. o-1			Alternate string 2; translation for second string in most recent non-ALTSTR tuple. Terminated by 00H.					
...			Etc.					
p			FFH - marks end of strings.					

5.2.6 Tuple Processing Recommendations

This standard requires that system software be carefully coded in order to prevent incompatibilities from one system to another. The following are some specific recommendations.

The routine that reads a given tuple should be coded to start by examining the tuple code. If the tuple code is not recognized by the routine (e.g. if the code is vendor specific or represents an extension under a future standard), then the tuple should be ignored. If the code is not recognized, it is safe to read the code byte and the link byte. **However, other bytes within the tuple may represent active registers.**

5.2.6.1 If the tuple code is known, and if the tuple does not contain active registers (which is the case for all standard tuples), then the routine should copy bytes into a buffer in main storage. Bytes should be copied from the code byte up to the last byte before the next tuple. If the link field is FFH (meaning end-of-list) then a maximum of 256 bytes — the code byte, the link byte and 254 byte of potential tuple data — should be copied from the card to the main store.

5.2.6.2 When processing a long-link tuple, software should merely record the target address and address space. The software should not validate the target address, nor should it immediately begin processing of tuples from the target address. Similarly, when a no-link tuple is found, that fact should be recorded for later use.

Long-link and no-link tuples should be processed *after* reaching the end of the tuple chain. At that time, if a long-link is to be processed, software should validate the target address (by checking for a link-target tuple) and begin processing the target chain if it appears to be valid.

5.2.6.3 A long-link that points to an invalid tuple chain should not usually cause any diagnostic messages to be displayed to the user. This situation may result from an uninitialized card, from a card which was initialized for some unanticipated use, or from corrupted data. Since only the corrupted-data case merits a diagnostic message, it is better to assume either that the card is uninitialized, or that it is initialized in some non-conforming way.

5.2.7 Basic Compatibility Tuples

5.2.7.1 CISTPL_DEVICE, CISTPL_DEVICE_A: The Device Information Tuples

The device-information tuples contain information about the card's devices. The tuples contain: device speed, device size, device type, and address-space layout information for either Attribute-Memory or Common-Memory space. These are determined by the tuple code. A device-information tuple for Common-Memory space, [CISTPL_DEVICE, 01H], must be the first tuple in Attribute Memory. The device-information tuple for Attribute Memory is optional. See Table 5-10.

Table 5-10: Device Information Tuples

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_DEVICE (01H) or CISTPL_DEVICE_A (17H)					
1	TPL_LINK		Link to next tuple (at least m-1)					
			Device Info 1 (2 or more bytes)					
			Device Info 2 (2 or more bytes)					
...			(etc.) Device Info n (2 or more bytes)					
m			FFH (marks end of device info field)					

The tuple code CISTPL_DEVICE indicates that this tuple describes Common-Memory space. The code CISTPL_DEVICE_A indicates that this tuple describes Attribute-Memory space.

5.2.7.1.1 The Device Info Field

The device-information tuples are composed of a sequence of device info fields. Each info field is further composed of two variable-length byte sequences — the device ID and the device size. Each info field defines the characteristics of a group of addresses in the appropriate memory space.

5.2.7.1.2 Device ID

The device ID indicates the device type and the access time for a block of memory. See Table 5-11.

Table 5-11: Device ID

Byte	7	6	5	4	3	2	1	0
0	Device Type Code				WPS	Device Speed		
1	Extended Device Speed (if Device Speed Code equals 7H, otherwise omitted)							
2 .. m-1	Additional Extended Device Speed (if bit 7 of Extended Device Speed is 1, otherwise omitted)							
m .. n	Extended Device Type (if Device Type Code equals EH, otherwise omitted).							

The WPS bit indicates whether the Write Protect Switch is in control of the device(s) in this address range. When the bit is 0, the Write Protect Switch and WP signal indicate whether or not the device(s) is(are) writable. When the bit is 1, the device is always writable unless the device code is DTYPE_ROM, in which case this address range is never writable. See Section 4.8.9: Write Protect Function.

5.2.7.1.3 Device Speed Field

If the device speed/type byte is 00H, the effect is unspecified. If the device-size information is valid, the address range shall be treated as a NULL device. If the device speed/type byte is FFH, it shall be treated as an end marker.

Bits 0 through 2, and up to one or two additional bytes, represent the speed of the devices associated with this part of the address space. The speed value indicates the external card access time to this address range. See Section 4.9 for card-level timing. The device-speed field contains one of the following values:

Table 5-12: Device Speed Codes

Code	Name	Meaning
0H	DSPEED_NULL	Use when device type=null
1H	DSPEED_250NS	250 nsec
2H	DSPEED_200NS	200 nsec
3H	DSPEED_150NS	150 nsec
4H	DSPEED_100NS	100 nsec
5H-6H		(Reserved)
7H	DSPEED_EXT	Use extended speed byte.

The extended speed byte has the following layout:

7	6	5	4	3	2	1	0
EXT	Speed Mantissa				Speed Exponent		

If the extended speed byte is zero, then the byte should be ignored.

The EXT bit, if set, indicates that an additional extended-speed byte follows. The meaning of that byte is not presently defined. However, the string of extended-speed bytes may be arbitrarily long. It extends through (and includes) the first byte with bit 7 reset.

The extended-device speed *mantissa* and *exponent* specify the speed of the device, as follows:

Table 5-13: Extended Device Speed Codes

Mantissa		Exponent part	
Code	**Meaning**	**Code**	**Meaning**
0_H	Reserved	0_H	1 ns
1_H	1.0	1_H	10 ns
2_H	1.2	2_H	100 ns
3_H	1.3	3_H	1 µs
4_H	1.5	4_H	10 µs
5_H	2.0	5_H	100 µs
6_H	2.5	6_H	1 ms
7_H	3.0	7_H	10 ms
8_H	3.5		
9_H	4.0		
A_H	4.5		
B_H	5.0		
C_H	5.5		
D_H	6.0		
E_H	7.0		
F_H	8.0		

5.2.7.1.4 Device Type Field

Bits 4 through 7 of byte 0 of the device-speed/ID sequence indicate the device type. The following device types are defined:

Table 5-14: Device Type Codes

Code	Name	Meaning
0	DTYPE_NULL	No device. Generally used to designate a hole in the address space. If used, speed field should be set to 0_H.
1	DTYPE_ROM	Masked ROM
2	DTYPE_OTPROM	One-time programmable PROM
3	DTYPE_EPROM	UV EPROM
4	DTYPE_EEPROM	EEPROM
5	DTYPE_FLASH	Flash EPROM
6	DTYPE_SRAM	Static RAM (JEIDA has Nonvolatile RAM)
7	DTYPE_DRAM	Dynamic RAM (JEIDA has Volatile RAM)
8-C_H		Reserved
D_H	DTYPE_FUNCSPEC*	Function-specific memory address range. Includes memory-mapped I/O registers, dual-ported memory, communication buffers, etc., which are not intended to be used as general-purpose memory.
E_H	DTYPE_EXTEND	Extended type follows.
F_H		Reserved

*Notes

1. Device Type Codes are used to describe only devices which are fixed in their memory address, not dynamically relocatable devices. Relocatable devices are described by the configuration tuples.
2. Accesses to function-specific address ranges, DTYPE_FUNCSPEC, may be configuration-dependent.

The extended-device type, if specified, is reserved for future use. Bit 7, if set, indicates that the next byte is also an extended-type byte. The chain of extended-type bytes can continue indefinitely. The end is marked by an extended-device-type byte with bit 7 reset.

5.2.7.1.5 The Device Size Byte

Within a device-ID structure, following the device-speed/type information, is a device- size byte.

Table 5-15: Device Size Byte

7	6	5	4	3	2	1	0
# of address units - 1					Size Code		

Code	Units	Max Size
0	512 bytes	16K
1	2K	64K
2	8K	256K
3	32K	1M
4	128K	4M
5	512K	16M
6	2M	64M
7	Reserved	Reserved

Bits 3 through 7 represent the number of address units. A code of zero indicates 1 unit; a code of 1 indicates 2 units; and so on.

If the device-size byte is FFH, then this entry should be treated as an end marker for the device-information tuple. The device-type and speed information encoded for this entry should be ignored.

5.2.7.2 CISTPL_VERS_1: The Level 1 Version/ Product Information Tuple

This tuple contains Level-1-version compliance and card-manufacturer information. It applies to Level-1 tuples only. It should appear only once in each partition-descriptor chain. See Table 5-16.

Table 5-16: Level 1 Version / Product Information Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_VERS_1 (15H).					
1	TPL_LINK		Link to next tuple (at least m-1).					
2	TPLLV1_MAJOR		Major version number (04H)					
3	TPLLV1_MINOR		Minor version number (01H) for Release 2.0 and 2.01.					
4	TPLLV1_INFO		Product information string: name of the manufacturer, terminated by 00H. Name of product, terminated by 00H. Additional product information, in text; terminated by 00H. Suggested use: lot number. Additional product information, in text; terminated by 00H. Suggested use: programming conditions.					
m			FFH: marks end of list.					

5.2.7.3 CISTPL_JEDEC_C, CISTPL_JEDEC_A: The JEDEC Identifier Tuples

This optional tuple is provided for cards containing programmable devices. It provides an array of *k* entries, where *k* is the number of distinct entries in the device information tuple (codes 01H or 17H). There is a one-to-one correspondence between

JEDEC identifier entries in this tuple and device-information entries in the device-information tuple. See Table 5-17.

Table 5-17: The JEDEC Identifier Tuples

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Format tuple code (CISTPL_JEDEC_C,18H, or CISTBL_JEDEC _A,19H).					
1	TPL_LINK		Link to next tuple (at least m-1).					
2.3			JEDEC identifier for first device-info entry.					
4.5			JEDEC identifier for second device-info entry (if needed).					
6.m			JEDEC identifiers for remaining device-info entries (if needed).					

The TPL_CODE field indicates to which device-information tuple the JEDEC-identifier tuple corresponds. If the value is 18H [CISTPL_JEDEC_C], this tuple corresponds to the Common-Memory device-information tuple [CISTPL_DEVICE]. If the value is 19H [CISTPL_JEDEC_A], this tuple corresponds to the Attribute-Memory device-information tuple [CISTPL_DEVICE_A].

JEDEC identifiers consist of two bytes. The first byte is the device-manufacturer ID, as assigned by JEDEC committee JC-42.4. This byte, if valid, must always have an odd number of bits set true. The MSB (bit 7) of the byte is used as a parity bit to ensure that this constraint is met. The values of 0 and FFH are illegal JEDEC identifiers (due to their even parity). The value 0H indicates "no JEDEC identifier for this device."

The second byte contains manufacturer-specific information representing device type, programming algorithm, and the like. If the manufacturer ID is 00H, then this byte is reserved and should be set to zero.

JEDEC identifiers will only be provided for device-info entries that indicate some kind of programmable device. For all other entries, the corresponding JEDEC identifier field shall be absent, i.e. set to 00H.

Examples:

If a card consists of four "Company X" 27C512 devices, whose JEDEC identifier is (hypothetically) manufacturer: 40H, device ID:15H, then the header block might be laid out as follows:

```
/reg[0]:      {CISTPL_DEVICE,
/reg[2]:      /*link*/                 3,
/reg[4]:      /*type/speed*/           DTYPE_OTPROM | DSPEED_250NS,
/reg[6]:      /*size: units/code*/     ((1<<3) | 4)
/reg[8]:      /*end of tuple*/         FFH
              );

/reg[10]:     {CISTPL_JEDEC_C,         /*18h*/
/reg[12]:     /*link*/                 0FFH, /*(end of list)*/
/reg[14]:     /*manufacturer ID*/      40H,
/reg[16]:     /*mfr's info*/           15H,
              );
```

[Note: In the above example, the size byte indicates that 2 x 128K bytes are available, for a total of 256K. Programming software would use the JEDEC information to determine that in fact 4, 64K x 8 devices are present. Since JEIDA V4 /PCMCIA cards are always 16-bits wide, programming software could therefore deduce the organization of the card.]

The following tuples might be used to describe a card with 256K of OTPROM and 16 Kbytes of RAM. In this example, RAM occupies locations [0 - 03FFFH], and ROM occupies locations [20000H - 5FFFFH] (for ease of decoding).

```
/reg[0]:       (CISTPL_DEVICE,
/reg[2]:       /*link*/               7,
/reg[4]:       /*type/speed*/         DTYPE_SRAM | DSPEED_100NS,
/reg[6]:       /*size: units/code*/   ((1<<3) | 2)
/reg[8]:       /*type/speed*/         DTYPE_NULL | DSPEED_NONE,
/reg[10]:      /*size: units/code*/   ((0DH<<3) | 2)
/reg[12]:      /*type/speed*/         DTYPE_OTPROM | DSPEED_250NS,
/reg[14]:      /*size: units/code*/   ((1<<3) | 4)
/reg[16]:      /*end of tuple*/       FFH
               );

/reg[18]:      (CISTPL_JEDEC_C,
/reg[20]:      /*link*/               0FFH, /*(end of list)*/
/reg[22]:      /*RAM: no code*/       0,
/reg[24]:      /*no info*/            0,
/reg[26]:      /*hole: no code*/      0,
/reg[28]:      /*no info*/            0,
/reg[30]:      /*manufacturer ID*/    40H,
/reg[32]:      /*mfr's info*/         15H,
               );
```

[Note: Place holders were left in the CISTPL_JEDEC tuple corresponding to the RAM and hole entries in the device tuple.

5.2.7.4 CISTPL_DEVICE_OC, CISTPL_DEVICE_OA: The Other Conditions Device Information Tuples

The Other Conditions device-information tuples contain information about the card's devices under a non-default set of operating conditions. The tuples are identical in format to the device-information Common and Attribute Memory tuples except that an Other-Conditions-information field precedes the first device-info field. There may be several CISTPL_DEVICE_OC and CISTPL_DEVICE_OA tuples in the CIS — one to describe the card under each set of alternative operating conditions.

There must be a one- to-one correspondence between entries in the CISPTL_DEVICE tuple and each CISTPL_DEVICE_OC tuple, as well as between the entries in the CISTPL_DEVICE_A tuple and each CISTPLE_DEVICE_OA tuple. Null devices are counted in the matching process. See Table 5-18.

Table 5-18: Other Conditions Device Information Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_DEVICE_OC (1CH) and CISTPL_DEVICE_OA (1DH).					
1	TPL_LINK		Link to next tuple (at least m-1).					
			Other Conditions Info (1 or more bytes)					
			Device Info 1 (2 or more bytes)					
			Device Info 2 (2 or more bytes)					
...			(etc.) Device Info n (2 or more bytes)					
m			FFH (marks end of device info field).					

The Other-Conditions-information field is a sequence of one or more bytes each with seven bits of conditions and one extension bit. The condition bits indicate the set of defined conditions which apply to the device information in the tuple. There are presently two condition bits defined — the MWAIT bit and the 3VCC bit. All undefined bits are reserved for future standardization and must be zero. Bit 7 of each byte is the extension bit which indicates that another byte of conditions follows the present byte. See Table 5-16.

Table 5-19: Other Conditions Information Field

Byte	7	6	5	4	3	2	1	0
0	EXT	Reserved, must be 0					3VCC	MWAIT
1..n	Additional Bytes		Each is present only if EXT bit is set in previous byte					

The 3VCC bit, when set to logic 1, indicates the card is a Dual Operating Voltage card, and the tuple defines the device characteristics at 3.3 volt operation. When this bit is set to logic 0, it defines device characteristics at other than 3.3 volts Vcc. The default (5 volt) operating conditions apply unless modified by another (yet to be defined) condition bit.

The MWAIT bit, when set to logic 1, indicates the device information applies to the minimum cycle with WAIT states. When logic 0, the MWAIT bit indicates device information applies when the wait signal is ignored by the host.

When both the 3VCC bit and MWAIT bit are logic 1, the tuple defines the minimum memory access when using WAIT at 3.3 volts Vcc.

If all the condition bits are zero, the tuple is invalid, as these cases would duplicate the CISTPL_DEVICE and CISTPL_DEVICE_A tuples.

5.2.7.5 Manufacturer Identification Tuple

The manufacturer identification tuple contains information about the manufacturer of a PC Card. Two types of information are provided: the PC Card's manufacturer and a manufacturer card number. Only one manufacturer identification tuple may be present in the card information structure.

Byte/Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	*TPL_CODE*		*CISTPL_MANFID (20H)*					
1	*TPL_LINK*		*Link to next tuple (at least 4)*					
2..3	*TPLMID_MANF*		*PCMCIA PC Card manufacturer code*					
4..5	*TPLMID_CARD*		*Manufacturer information (Part Number and/or Revision)*					

*The **TPLMID_MANF** field identifies the PC Card's manufacturer. The code is assigned by PCMCIA. The first 256 identifiers (0000H through 00FFH) are reserved for manufacturers who have JEDEC IDs assigned by JEDEC Publication 106. Manufacturers with JEDEC IDs may use their eight-bit JEDEC manufacturer code as the least significant eight bits of their PCMCIA PC Card manufacturer code. In this case, the most significant eight bits must be zero (0). For example, if a JEDEC manufacturer code is 089H, their PCMCIA PC Card manufacturer code is 0089H.*

*The **TPLMID_CARD** field is reserved for use by the PC Card's manufacturer. It is anticipated that the field will be used to store card identifier and revision information.*

5.2.7.6 Function Identification Tuple

The function identification tuple contains information about the functionality provided by a PC Card. Information is also provided to enable system utilities to decide if the PC Card should be automatically configured during system initialization. If the PC Card provides more than one function, more than one function identification tuple is present. If additional function-specific information is available, one or more function extension tuples will follow this tuple.

Byte/Bit	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_FUNCID (21$_H$)					
1	TPL_LINK		Link to next tuple (at least 2)					
2	TPLFID_FUNCTION		PC Card function code (see Table 5-19a)					
3	TPLFID_SYSINIT		System initialization bit mask (see Table 5-19b)					

*The **TPLFID_FUNCTION** field identifies the function provided by the PC Card. If the PC Card provides multiple functions, multiple function identification tuples will be present. The first function identification tuple identifies the PC Card as a multi-function card and additional function identification tuples are present for each function provided.*

As noted above, additional information about a specific function may be available in function extension tuples. These extension tuples follow their related function identification tuple. On multi-function cards, the first function identification tuple indicates the PC Card has multiple functions with a PC Card function code of zero (0). The next function identification tuple identifies the first function provided by the card. If additional information about that function is available, one or more function extension tuples specific to the function follow. For each additional function present on the card, an additional function identification tuple is present, optionally followed by its function extension tuples.

The defined function codes are listed in Table 5-19a: PC Card Functions.

Table 5-19a : PC Card Functions

Code	Name	Meaning
0	Multi-Function	PC Card has multiple functions, see following function identification tuples for individual functions
1	Memory	Memory Card (RAM, ROM, EPROM, flash, etcetera)
2	Serial Port	Serial I/O port, includes modem cards
3	Parallel Port	Parallel printer port, may be bi-directional
4	Fixed Disk	Fixed drive, may be silicon may be removable
5	Video Adapter	Video interface, extension tuples identify type and resolutions
6	NetworkLAN Adapter	Local Area Network adapter
7	AIMS	Auto Incrementing Mass Storage card
8..FFh	Reserved	Unused in this release. Reserved by PCMCIA for future use.

*The **TPLFID_SYSINIT** is a bit-mapped field. It allows PC Cards which supply standard system resources to be installed during system initialization. The bit definitions are described in Table 5-19b: System Initialization Byte.*

Table 5-19b : System Initialization Byte

7	6	5	4	3	2	1	0
Reserved for future use, must be set to zero (0)						ROM	POST

*The **POST** bit indicates that the Power-On Self Test routines should attempt to configure the PC Card during system initialization. The POST routines in the system initialization code may attempt to resolve resource conflicts. How this is handled is implementation-specific within the host system.*

*The **ROM** bit indicates the PC Card contains a system expansion ROM which should also be installed during system initialization. Information describing the ROM to be installed is in the **TPCE_FS Feature Selection** Byte (see section 5.2.8.3.4.) and related **Memory Space Description Structure** (see section 5.2.3.8.9.).*

Function identification tuples indicating a multi-function card (a function code of zero) require special handling for the System Initialization Byte. In this case, the System Initialization Byte is the logical OR of the System Initialization Bytes for all the specific function identification tuples that follow the initial function identification tuple indicating the PC Card contains multiple functions. This allows system initialization code in the host computer to quickly determine whether any further processing of the Card Information Structure is required to configure a PC Card or install an expansion ROM.

Only the specific function requiring POST or BOOT processing should have the appropriate bits set in the System Initialization Byte. (With the exception of the multi-function function identification tuple, as noted above).

5.2.7.7 Function Extension Tuple

The function extension tuple contains information about a specific PC Card function. The information provided is determined by the function identification tuple that is being extended. Each function has a defined set of extension tuples. They are described in subsections following this one.

Byte/Bit	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_FUNCE (22H)					
1	TPL_LINK		Link to next tuple (see following sections)					
2	TPLFE_TYPE		Type of extended data (see following sections)					
3..n	TPLFE_DATA		Function information (see following sections)					

*The **TPLFE_TYPE** field identifies the type of extended information provided about a function by this tuple. This information varies depending on the function being extended.*

*The **TPLFE_DATA** field is the specific information being provided by the extended function. The content of this field varies depending on the function being extended and the **TPLFE_TYPE** field.*

Each of the function classes identified in Table 5-19a: PC Card Functions may be extended. Function extension tuples are optional. If present, they relate to the function identification tuple preceding them in the Card Information Structure.

Actual card identification extensions are to be determined by working groups concerned with specific PC Card function classes. The following subsections describe specific extension tuples. Multi-function function identification tuples are not extendable. The subsections are numbered corresponding to the function code being extended. If a particular function may be extended with multiple types of extended data, a further division is provided for each type of extension data within the subsection.

5.2.7.7.1 Modem Function Extension Tuples

The tuples defined in this document contain information which describe the operational features of a modem. Modems are identified using a PCMCIA Card Function Id of a Serial Port and defined further using CISTPL_FUNCID Extension Tuples to describe specific capabilities. The structure of the tuples are determined by a code appearing in the TPLFE_TYPE field (See section 5.2.7.7). CISTPL_FUNCID Extension Tuples are intended for informational use and not configuration control. Only those features commonly available and of particular interest to application software are included.

The TPLFE_TYPE field presented below, distinguishes each successive CISTPL_FUNCE serial port/modem extension tuple. Since all CISTPL_FUNCE extension tuples contain the same code (22H), both the position and the value of the TPLFE_TYPE field must be used to identify the functionality described by a particular extension tuple. Consequently, the serial port/modem extension tuples defined in this document must always appear immediately after the CISTPL_FUNCE tuple defined for serial port devices.

The codes defined for the TPLFE_TYPE field are presented below. Please note that separate codes are provided to describe the modem and serial port interfaces for individual modem services. Separate codes are also provided to describe the modem and serial port interfaces common to all modem services.

Table 5-19c: TPLFE_TYPE: Extension Tuple Type Codes

7	6	5	4	3	2	1	0
PC Card Subfunction Descriptor				*PC Card Subfunction Id*			

PC Card Subfunction Id: *Identifies a category of services provided by the modem.*

Code	Description
0	*Identifies the extension tuple as a description of a serial port interface.*
1	*Identifies the extension tuple as a description of the capabilities of the modem interface,* ***common to all modem services.***
2	*Identifies the extension tuple as a description of* ***data*** *modem services.*
3	*Identifies the extension tuple as a description of* ***facsimile*** *modem services.*
4	*Identifies the extension tuple as a description of* ***voice*** *encoding services.*
5	*Identifies the extension tuple as a description of the capabilities of the* ***data*** *modem interface.*
6	*Identifies the extension tuple as a description of the capabilities of the* ***facsimile*** *modem interface.*
7	*Identifies the extension tuple as a description of the capabilities of the* ***voice*** *modem interface.*
8	*Identifies the extension tuple as a description of a serial port interface for* ***data*** *modem services.*
9	*Identifies the extension tuple as a description of a serial port interface for* ***facsimile*** *modem services.*
10	*Identifies the extension tuple as a description of a serial port interface for* ***voice*** *modem services.*
11-15	*Reserved for future standardization, set to zero.*

PC Card Subfunction Descriptor: *Identifies a subcategory of services provided by the modem.*

Code	Description
1-15	*Identifies the extension tuple as a description of the EIA/TIA Service Class specified in the numeric value of the code.*

5.2.7.7.1.1 Serial Port Interface Function Extension

This tuple indicates the type and capabilities of the serial port interface used in communicating with the modem. A specific code in the TPLFE_TYPE field is defined to describe the serial port interface to all modem services. Additional codes are also defined to describe the serial port interface for each available modem service (data, facsimile, and/or voice). Please refer to the Table 5-19.c for a list of all TPLFE_TYPE field codes.

The UART is identified with a generic reference to the de-facto standard it conforms to (i.e. 16450, 16550) not, however, a specific product identification. The UART capabilities field is provided to describe the asynchronous data formats supported in UART implementations which vary from the standard. When describing a de-facto standard UART the contents of this field may be set to zero.

The structure of the tuple and the codes currently defined for each field are presented below.

Table 5-19d: Serial Port Interface Function Extension Tuple

Byte/Bit	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_FUNCE (22H)					
1	TPL_LINK		Link to next tuple (04H minimum)					
2	TPLFE_TYPE		Type of extended data (0(0,8,9,A)H), see Table 5-19.c)					
3	TPLFE_UA		Identification of the type of UART in use.					
4-5	TPLFE_UC		Identification of the UART capabilities.					

Table 5-19e: TPLFE_UA: UART Identification Field

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.			interface				

Interface: The type of the UART is described using the codes defined below.	
Code	Description
0	Indicates that an 8250 UART is present.
1	Indicates that a 16450 UART is present.
2	Indicates that a 16550 UART is present.
3-31	Reserved for future standardization, set to zero.

Table 5-19f: TPLFE_UC: UART Capabilities Field

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.				**even parity**	**odd parity**	**mark parity**	**space parity**
reserved (set to zero)	**2 stop bits**	**1.5 stop bits**	**1 stop bit**	**8 bit char**	**7 bit char**	**6 bit char**	**5 bit char**

space parity: The protocol where the parity bit is always reset.	
Code	Description
0	Indicates that space parity is not supported.
1	Indicates that space parity is supported.

mark parity: The protocol where the parity bit is always set.	
Code	Description
0	Indicates that mark parity is not supported.
1	Indicates that mark parity is supported.

odd parity: The protocol where the parity bit is generated to create an odd number of set bits.

Code	Description
0	Indicates that odd parity is not supported.
1	Indicates that odd parity is supported.

even parity: The protocol where the parity bit is generated to create an even number of set bits.

Code	Description
0	Indicates that even parity is not supported.
1	Indicates that even parity is supported.

5 bit char: The encoding of data where each serial data unit contains 5 data bits.

Code	Description
0	Indicates that the 5 bit data format is not supported.
1	Indicates that the 5 bit data format is supported.

6 bit char: The encoding of data where each serial data unit contains 6 data bits.

Code	Description
0	Indicates that the 6 bit data format is not supported.
1	Indicates that the 6 bit data format is supported.

7 bit char: The encoding of data where each serial data unit contains 7 data bits.

Code	Description
0	Indicates that the 7 bit data format is not supported.
1	Indicates that the 7 bit data format is supported.

8 bit char: The encoding of data where each serial data unit contains 8 data bits.

Code	Description
0	Indicates that the 8 bit data format is not supported.
1	Indicates that the 8 bit data format is supported.

1 stop bit: The number of stop bits encoded in each serial data unit.

Code	Description
0	Indicates that the use of 1 stop bit is not supported.
1	Indicates that the use of 1 stop bit is supported.

1.5 stop bit: The number of stop bits encoded in each serial data unit.

Code	Description
0	Indicates that the use of 1.5 stop bits is not supported.
1	Indicates that the use of 1.5 stop bits is supported.

2 stop bit: The number of stop bits encoded in each serial data unit.

Code	Description
0	Indicates that the use of 2 stop bits is not supported.
1	Indicates that the use of 2 stop bits is supported.

5.2.7.7.1.2 Modem Interface Function Extension

This structure describes the characteristics of the modem interface when considered separately from the UART. This includes an indication of the types of flow control supported, the size of the command buffer, DCE to DTE buffer, and the DTE to DCE buffer.

A specific code in the TPLFE_TYPE field is defined to describe the modem interface to all modem services. Additional codes are also defined to describe the modem interface for each available modem service (data, facsimile, and/or voice). Please refer to the Table 5-19.c for a list of all TPLFE_TYPE field codes.

Table 5-19g: Modem Interface Function Extension Tuple

Byte/Bit	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_FUNCE (22h)					
1	TPL_LINK		Link to next tuple (09h minimum)					
2	TPLFE_TYPE		Type of extended data (0{1,5,6,7}h), see Table 5-19.c)					
3	TPLFE_FC		Supported flow control methods.					
4	TPLFE_CB		Size in bytes of the DCE command buffer.					
5-7	TPLFE_EB		Size in bytes of the DCE to DTE buffer.					
8-10	TPLFE_TB		Size in bytes of the DTE to DCE buffer.					

Table 5-19h: TPLFE_FC: Flow Control Methods

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.			**transparent**	**rx h/w flow ctrl**	**tx h/w flow ctrl**	**rx xon/xoff**	**tx xon/xoff**

tx xon/xoff: DTE to DCE transmit flow control using DC1/DC3 for XON/XOFF	
Code	Description
0	Indicates that XON/XOFF DTE to DCE transmit flow control is not supported.
1	Indicates that XON/XOFF DTE to DCE transmit flow control is supported.

rx xon/xoff: DCE to DTE receive flow control using DC1/DC3 for XON/XOFF	
Code	Description
0	Indicates that XON/XOFF DCE to DTE receive flow control is not supported.
1	Indicates that XON/XOFF DCE to DTE receive flow control is supported.

tx h/w flow ctrl: DTE to DCE transmit flow control (CTS)	
Code	Description
0	Indicates that hardware based DTE to DCE transmit flow control is not supported.
1	Indicates that hardware based DTE to DCE transmit flow control is supported.

rx h/w flow ctrl: DCE to DTE receive flow control (RTS)	
Code	Description
0	Indicates that hardware based DCE to DTE receive flow control is not supported.
1	Indicates that hardware based DCE to DTE receive flow control is supported.

transparent: DCE to DCE flow control	
Code	Description
0	Indicates that transparent flow control is not supported.
1	Indicates that transparent flow control is supported.

Table 5-19i: TPLFE_CB: DCE Command Buffer

7	6	5	4	3	2	1	0
size of DCE command buffer							

size of DCE command buffer: The size of the command line buffer in bytes (v) where the field value is computed by v/4 - 1.	
Code	Description
v/4 - 1	Indicates the number of bytes within the command line buffer. To compute the actual size of the command buffer, the value of this field (represented in the formula as n) is increased by 1, then multiplied by 4 ((n+1)*4). A command buffer of 256 bytes is represented by a field value of 3fh (63).

Table 5-19j: TPLFE_EB: Modem DCE to DTE Buffer Field

7	6	5	4	3	2	1	0
size of DCE-DTE buffer -- LSB of LSW							
size of DCE-DTE buffer -- MSB of LSW							
size of DCE-DTE buffer -- LSB of MSW							

size of DCE to DTE buffer: The largest possible size in bytes of the modem's DCE to DTE buffer used in buffering received data, not including the UART FIFO buffer. The actual number of bytes in the buffer appears in this field to allow a more precise definition of its size.	
Code	Description
0	Indicates that no DCE to DTE buffer is present.
1-16777215	Indicates that a DCE to DTE buffer is available and its size in bytes.

Table 5-19k: TPLFE_TB: Modem DTE to DCE Buffer Field

7	6	5	4	3	2	1	0
size of DTE-DCE buffer -- LSB of LSW							
size of DTE-DCE buffer -- MSB of LSW							
size of DTE-DCE buffer -- LSB of MSW							

size of DTE to DCE buffer: The largest possible size in bytes of the modem's DTE to DCE buffer used in buffering transmitted data, not including the UART FIFO buffer. The actual number of bytes in the buffer appears in this field to allow a more precise definition of its size.	
Code	Description
0	Indicates that no DTE to DCE buffer is present.
1-16777215	Indicates that a DTE to DCE buffer is available and its size in bytes.

5.2.7.7.1.3 Data Modem Function Extension

This structure describes the capabilities of the data modem. This includes the level of error correction and data compression supported, the highest possible data rate supported in the DTE to UART interface of the data modem, the command set in use for DTE to DCE control and configuration (eg. AT command set), the CCITT standard and non CCITT standard modulations supported, the standardized data encryption methods supported, and the CCITT defined country code of the target market The escape codes supported for the return to command mode are also described. Features which cannot be included in the categories reserved for the capabilities mentioned above are included in the "Miscellaneous End User Feature Selection" field.

The CCITT country code field defines the countries targeted for distribution of the data modem. This field has been specified to support a maximum value of 254 (255 or FFh is reserved as a country code list terminator, see below) in accordance with the CCITT standard T.35 which has assigned each member country or area a unique 8 bit value. Please refer to T.35 Annex A for a list of country or area codes currently assigned.

To specify support of multiple countries, the country code field TPLFE_CD was positioned as the last field in the Data Modem Function Extension Tuple. Additional country codes may be specified by increasing the value of the TPL_LINK field by one for each additional country specified. A value of FFh is used to indicate the end of the country code list if it does not extend to the end of the tuple. Any additional fields which follow this value must be represented by an appropriate increase in the value of the TPL_LINK field and are considered a description of manufacturer specific capabilities.

Table 5-19l: Data Modem Function Extension Tuple

Byte/Bit	7	6	5	4	3	2	1	0
0	**TPL_CODE**		CISTPL_FUNCE (22h)					
1	**TPL_LINK**		Link to next tuple (0Ch minimum)					
2	**TPLFE_TYPE**		Type of extended data (02h, see Table 5-19.c)					
3-4	**TPLFE_UD**		The highest possible bit rate in the DTE to UART interface.					
5-6	**TPLFE_MS**		Modulation standards supported					
7	**TPLFE_EM**		Error correction/detection protocols and non CCITT modulation schemes supported					
8	**TPLFE_DC**		Data compression protocols supported					
9	**TPLFE_CM**		Command protocols supported					
10	**TPLFE_EX**		Indication of the escape mechanisms supported					
11	**TPLFE_DY**		Standardized data encryption					
12	**TPLFE_EF**		Miscellaneous end user feature selection					
13..n	**TPLFE_CD**		The CCITT defined country code (CCITT T.35/T.35 Annex A)					

Table 5-19m: TPLFE_UD: UART Interface Field

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.					**DTE to UART maximum data rate – MSB**		
DTE to UART maximum data rate – LSB							

DTE to UART maximum data rate: The highest bit rate supported for communications with the DTE (v) in increments of 75bps, where the field value is computed by v/75.	
Code	Description
v/75	Indicates the highest DTE to UART bit rate supported. To compute the actual bit rate, the value of this field (represented in the formula as n) is multiplied by 75 (n * 75). A bit rate of 115,200 would be represented by a field value of 600h (1536).

Table 5-19n: TPLFE_MS: Modulation Standards Field7

7	6	5	4	3	2	1	0
V.26bis	**V.26**	**V.22bis**	**Bell 212A**	**V.22A&B**	**V.23**	**V.21**	**Bell 103**
Reserved for future standardization, set to zero.		**VFAST**	**V.32bis**	**V.32**	**V.29**	**V.27bis**	**VFAST**

Bell 103:	300-bps duplex modem standardized for use in the GSTN.
V.21:	300-bps duplex modem standardized for use in the GSTN.
V.23:	600/1200-baud modem standardized for use in the GSTN.
V.22A&B:	1200-bps duplex modem standardized for use in the GSTN and for telephone-type leased circuits.
Bell 212A:	2400-bps duplex modem standardized for use in the GSTN.
V.22bis:	2400-bps duplex modem using the frequency-division technique standardized for use on the GSTN and on point-to-point two-wire leased telephone-type circuits.
V.26:	2400-bps modem standardized for use on 4-wire leased telephone-type circuits.
V.26bis:	2400-bps duplex modem standardized for use in the GSTN, specifying two alternate encoding schemes (A or B).
V.27bis:	4800/2400-bps modem with automatic equalizer standardized for use on leased telephone-type circuits.
V.29:	9600/7200/4800-bps modem standardized for use on point-to-point 4-wire leased telephone-type circuits
V.32:	A family of two-wire, duplex modems operating at data rates of up to 9600-bps for use on the GSTN or on lease telephone-type circuits.
V.32bis:	A family of two-wire, duplex modems operating at data rates of up 14.4-Kbps for use on the GSTN or on lease telephone-type circuits.
VFAST:	A proposed standard supporting data rates as high as 28,800-bps.
Code	Description
0	Indicates that the specified standard is not supported.
1	Indicates that support of the specified standard is provided.

Table 5-19t: TPLFE_EF: Miscellaneous End User Feature Selection

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.							Caller Id

Caller Id: A protocol which defines the encoding of specific information to allow the answering station to identify the calling station.	
Code	Description
0	Indicates that caller id decoding services are not provided.
1	Indicates that caller id decoding services are provided.

Table 5-19u: TPLFE_CD CCITT Country Code

7	6	5	4	3	2	1	0
country code							

country code: CCITT numeric code assigned to the country targeted for distribution of this modem (Annex A, Recommendation T.35).	
Code	Description
0-254	The assigned CCITT country code.
255	Indicates the end of the variable length country code list if it does not extend to the end of the tuple. The fields which follow this field when a value of 255 is present represent manufacturer specific capabilities.

5.2.7.7.1.4 Document Facsimile Function Extension

The Document Facsimile Function Extension Tuple describes the capabilities of the facsimile modem. This includes the highest possible data rate in the DTE to UART interface of the facsimile modem, the CCITT modulation standards supported, the CCITT defined country code of the target market, the standardized data encryption methods supported, and the various document facsimile features supported.

A separate tuple shall be used to describe the capabilities of each supported Service Class. Thus a facsimile modem supporting the EIA/TIA-578 Service Class 1 standard and the draft standard Service Class 2 must include two separate extension tuples. The PC Card Subfunction Descriptor represented in the least significant nibble of the TPLFE_TYPE field would vary with each tuple.

The document facsimile services of Error Correction Mode, Voice Request, Polling, File Transfer, Password, Selective Polling, Character Mode, and Copy Quality were not included in this tuple. Many have not yet achieved wide spread use and are still in the midst of the CCITT approval process. Others require control and configuration through the use of manufacturer specific AT commands and are consequently not within the scope of this document.

The CCITT country code field defines the country targeted for distribution of the facsimile modem. This field has been specified to support a maximum value of 254 (255 or FFh is reserved as a country code list terminator, see below) in accordance with the CCITT standard T.35 which has assigned each member country or area a unique 8 bit value. Please refer to T.35 Annex A for a list of country or area codes currently assigned.

To specify support of multiple countries, the country code field TPLFE_CF is positioned as the last field in the Document Facsimile Function Extension Tuple. Additional country codes may be specified by increasing the value of the TPL_LINK field by one for each additional country specified. A value of FFh indicates the end of the country code list if it does not extend to the end of the tuple. Any additional fields which follow this value must be represented by an appropriate increase in the value of the TPL_LINK field and are considered a description of manufacturer specific capabilities.

Table 5-19z: TPLFE_FY Standardized Data Encryption

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.							

Table 5-19aa: TPLFE_FS Document Facsimile Feature Selection

7	6	5	4	3	2	1	0
pass- word	**file transfer**	**polling**	**voice request**	**error correc mode**	**T.6**	**T.4**	**T.3**
Reserved for future standardization, set to zero.							

T.3: Standardization of group 2 facsimile apparatus for document transmission.	
Code	Description
0	Indicates that T.3 is not supported in the specified Service Class.
1	Indicates that T.3 is supported in the specified Service Class.

T.4: Facsimile coding schemes and coding control functions for group 3 facsimile apparatus.	
Code	Description
0	Indicates that T.4 is not supported in the specified Service Class.
1	Indicates that T.4 is supported in the specified Service Class.

T.6: Facsimile coding schemes and coding control functions for group 4 facsimile apparatus.	
Code	Description
0	Indicates that T.6 is not supported in the specified Service Class.
1	Indicates that T.6 is supported in the specified Service Class.

error crctn. mode: A mode of operation allowing error correction of document facsimile page data.	
Code	Description
0	Indicates that ECM is not supported in the specified Service Class.
1	Indicates that ECM is supported in the specified Service Class.

voice request: Interruption of the transmission of a facsimile image to allow voice contact with the remote station.	
Code	Description
0	Indicates that voice request is not supported in the specified Service Class.
1	Indicates that voice request is supported in the specified Service Class.

polling: A request to receive or send a facsimile regardless of which station is considered the originator.	
Code	Description
0	Indicates that polling is not supported in the specified Service Class.
1	Indicates that polling is supported in the specified Service Class.

file transfer: CCITT defined procedure for transferring files between PC based facsimile DCE 's.	
Code	Description
0	Indicates that file transfer is not supported in the specified Service Class.
1	Indicates that file transfer is supported in the specified Service Class.

password: A mode of operation allowing secured facsimile transmissions.	
Code	Description
0	Indicates that password is not supported in the specified Service Class.
1	Indicates that password is supported in the specified Service Class.

Table 5-19ab: TPLFE_CF CCITT Country Code

7	6	5	4	3	2	1	0
country code							

country code: CCITT numeric code assigned to the country targeted for distribution of this modem (Annex A, Recommendation T.35).	
Code	Description
0-254	The assigned CCITT country code.
255	Indicates the end of the variable length country code list if it does not extend to the end of the tuple. The fields which follow this field when the value of 255 is present represent manufacturer specific capabilities.

5.2.7.7.1.5 Voice Function Extension

This structure describes the capabilities of a modem equipped to process digitally encoded voice data. The highest possible data rate in the DTE to UART interface of the voice modem is described. The PC Card Subfunction Descriptor specifies the number of the Service Class used to activate the voice command mode.

The sample rate, size, and compression methods supported are represented in a series of three separate tuples. Each tuple is repeated for the supported sample rates, sizes, and compression methods. These fields serve only to described the range of supported options not, however, the various combinations of each. As extended voice AT commands are defined by the EIA/TIA and the operation of a voice equipped modem is standardized this tuple will be appropriately updated.

Table 5-19ac: Voice Function Extension Tuple

Byte/Bit	7	6	5	4	3	2	1	0
0	TPL_CODE		CISTPL_FUNCE (22h)					
1	TPL_LINK		Link to next tuple (06h minimum)					
2	TPLFE_TYPE		Type of extended data (84h assuming Service Class 8, see Table 5-19.c)					
3-4	TPLFE_UV		The highest possible data rate in the DTE to UART interface.					
5..	TPLFE_SR		A variable length list of the sample rates supported.					
..	TPLFE_SS		A variable length list of the sample sizes supported.					
..n	TPLFE_SC		A variable length list of the EIA/TIA Compression Method Identifiers. **It is not yet certain if the EIA/TIA will maintain the assignment of these numeric Id's. Consequently, until the outcome is determined, if manufacturer specific data is present at the end of the tuple, this field shall remain set to zero.**					

Table 5-19ad: TPLFE_TYPE Type of CISTPL_FUNCE Extended Data

7	6	5	4	3	2	1	0
PC Card Subfunction Descriptor				PC Card Subfunction Id			

PC Card Subfunction Id:	
Code	Description
4	Indicates the presence of PCM voice encoding capabilities. See Table 5-19.c for a list of all Subfunction Codes defined.

PC Card Subfunction Descriptor:	
Code	Description
0	Indicates that voice encoding services are provided using a protocol other than that defined in the applicable Service Class.
4-15	The numeric value of the Service Class reserved for the definition of the extended voice AT commands **(currently being standardized)**. The actual value used by the manufacturer shall be indicated in this field.

Table 5-19ae: TPLFE_UV: UART Interface Field

7	6	5	4	3	2	1	0
Reserved for future standardization, set to zero.					**DTE to UART maximum data rate – MSB**		
DTE to UART maximum data rate – LSB							

DTE to UART maximum data rate: The highest bit rate supported for communications with the DTE (v) in increments of 75bps, where the field value is computed by v/75.

Code	Description
v/75	Indicates the highest DTE to UART bit rate supported. To compute the actual bit rate the value of this field (represented in the formula as n) is multiplied by 75 (n * 75). A bit rate of 115,200 would be represented by a field value of 600h (1536).

Table 5-19af: TPLFE_SR: Sample Rate Field

7	6	5	4	3	2	1	0
Resvd., set to zero	**Sample Rate in KHz**						
Resvd., set to zero	**Sample Rate in KHz/100**						

Sample Rate in KHz: The voice sampling rate supported in increments of a 1000 Hz. A value of 11.025 KHz would be represented in this field as 0Bh.

Code	Description
0	Indicates the end of the variable length list of supported sample rates. The next byte represents the beginning of the variable length list of supported sample sizes.
1-99	Indicates the sample rate supported as a multiple 1000 Hz. A value of 11.025 KHz is be represented in this field as 0Bh.

Sample Rate in KHz/100: The voice sampling rate supported in units of 1/100 KHz. The fractional component of 11.025 KHz is represented in this field as 02h. The final sample rate is derived by adding the fractional component in this field to the whole number value in the field Sample Rate in KHz. A value of 11.025 KHz would be derived by adding 0Bh KHz or 11 KHz to 02h/100 KHz or .02 KHz for a value of 11.02 KHz in four significant digits.

Code	Description
0-99	Indicates the fractional component of the supported sample rate in units of 1/100 KHz. A value of 11.025 KHz would be represented in this byte as 02h.

Table 5-19ag: TPLFE_SS: Sample Size Field

7	6	5	4	3	2	1	0
Reserved, set to zero			**Sample Size in Bits**				
Reserved, set to zero				**Sample Size in Bits/10**			

Sample Size in Bits: The sample size supported in bits.

Code	Description
0	Indicates the end of the variable length list of supported sample sizes. The next byte represents the beginning of the variable length list of supported data compression methods.
1-15	Indicates the supported sample size in units of 1 bit. A value of 2.66 bits is represented in this field as 02h.

Sample Size in Bits/10: The sample size supported in units of 1/10 bits.

Code	Description
0-9	Indicates the fractional component of the supported sample size in units of 1/10 bits. A value of 2.66 bits is represented in this field as 06h. The final sample size is derived by adding the fractional component in this field to the whole number value in the field Sample Size in Bits. A value of 2.66 bits would be derived by adding 02h or 2 bits to 06h or .6 bits for a value of 2.6 bits in 2 significant digits.

Table 5-19ah: TPLFE_SC: Voice Compression Methods

7	6	5	4	3	2	1	0
Voice Compression Method Id, (Reserved for future standardization, set to zero)							

Voice Compression Method Id: Assuming the EIA/TIA presides over the assignment of id's to the many compression methods currently available, this field represents the EIA/TIA assigned Compression Method Identifier.	
Code	Description
0	Indicates the end of the variable length list of supported Compression Method Identifiers. The next byte represents the beginning of fields containing manufacturer specific data. This byte need not be present if the list extends to the end of the tuple.
1-255	Indicates the EIA/TIA assigned Compression Method Identifier associated with the supported compression method.

5.2.7.7.2 Disk Device Function Extension Tuples

See the PC Card ATA Specification for Disk Device Function Extension Tuples. Additional function extension tuples may be added in the future for PC Card ATA and other disk interfaces.

5.2.7.8 Device Geometry Tuple

The device geometry info tuple CISTPL_DEVICEGEO is required for certain memory technologies (e.g. Flash Memory, EEPROM) or card-integrated memory subsystems. While conceptually similar to a DOS disk geometry tuple (CISTPL_GEOMETRY), it is not a format-dependent property; this deals with the fixed architecture of the memory device(s) or subsystem(s). Rather than being specific to a partition within a larger device type, this applies to the entire address range of that device type. Accordingly, each device entry in CISTPL_DEVICE or CISTPL_DEVICE_A, including null device space, must have a corresponding device geometry tuple (CISTPL_GEODEVICE) entry when this tuple is employed.

Byte	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	*TPL_CODE Tuple codes for the device geometry tuples.* *(CISTPL_DEVICEGEO, 1Eh, CISTPL_DEVICEGEO_A, 1Fh)*							
1	*TPL_LINK Link to next tuple (at least 6*k).*							
2 .. 7	*Device Geometry info for Device 1.*							
8 .. 13	*Device Geometry info for Device 2.*							
..	*(etc.)* *Device Geometry info for Device k.*							

5.2.7.8.1 Device Geometry Info Field

Device geometry info field have a 1:1 correspondence with their associated device info tuple fields. The codes for device geometry info are as follows:

Byte	7	6	5	4	3	2	1	0
1	DGTPL_BUS		Value = n, where system bus width = 2(n-1) bytes. N = 2 for standard PCMCIA Release 1.0/2.0 Cards. The value n = 0 is not allowed.					
2	DGTPL_EBS		Value = n, where memory array/subsystem's physical memory segments have a minimum erase block size of 2(n-1) address increments of DGTPL_BUS-wide accesses. The value n = 00h is not allowed.					
3	DGTPL_RBS		Value = n, where memory array/subsystem's physical memory segments have a minimum read block size of 2(n-1) address increments of DGTPL_BUS-wide accesses. The value n = 00h is not allowed.					
4	DGTPL_WBS		Value = n, where memory array/subsystem's physical memory segments have a minimum write block size of 2(n-1) address increments of DGTPL_BUS-wide accesses. The value n = 00h is not allowed.					
5	DGTPL_PART		Value = p, where memory array/subsystem's physical memory segments can have partitions subdividing the arrays in minimum granularity of 2(p-1) number of erase blocks. P = 1 where array partitioning at erase block boundaries is allowed. The value p = 00h is not allowed.					
6	DGTPL_HWIL		Value = q, where card architectures employ a multiple of 2(q-1) times interleaving of the entire memory arrays or subsystems with the above characteristics. Non-interleaved cards have values of q = 1. The value q = 00h is not allowed.					

*If the Device Geometry table is used, the entire six byte entry must be filled out for each entry in CISTPL_DEVICE or CISTPL_DEVICE_A, even for devices without any special geometries (e.g. ROM, RAM), so that a 1:1 correspondence between Device and Device Geometry tables of both COMMON and ATTRIBUTE regions exists. The table length is 6*k + 2 bytes, where k = the number of devices described. Bit 7 of the last (sixth) byte of each detailed device geometry info flags extended device geometry info (format to be determined) for that particular device.*

The DGTPL_BUS tuple has a value of n = 2 for 16-bit bus width of PCMCIA Release 1.0 and 2.0 cards regardless of word- or byte-wide operation (byte-wide operation is logically low- and high-byte sequencing of a fundamental 16-bit-wide card-internal bus). The purpose of this entry is to accommodate possible wider-width PCMCIA cards of the future and/or to allow file systems to use this tuple structure in non-PCMCIA environments (e.g. system-resident memory arrays using the same file system).

NOTE:

The device geometry info is normalized to byte-equivalent geometry to simplify the context for the O/S- or driver-level software which utilizes it. Subsystems using internal bus widths less than 8-bits wide must employ low-level drivers which accept 8-bit minimum data from the higher levels.

The DGTPL_EBS, DGTPL_RBS, and DGTPL_WBS (address increment- or bus operation-based values) are multiplicative of the DGTPL_BUS entry (denoting bus width) to define the non-interleaved physical memory erase-, read-, and write-block sizes in bytes, respectively. The DGTPL_HWIL value for cards employing hardware-interleaved (i.e "banks" *of) memory arrays or subsystems (where DGTPL_HWIL _ 2) is multiplicative of the resulting non-interleaved erase-, read-, and write geometries. The product of these three geometry info layers yields the resulting card-level minimum physical block geometries.*

DGTPL_PART is a special partitioning info field based on physically distinct segments of the memory array(s) such that data are not affected by read/write/erase operations in adjacent partitions. It is multiplicative of the resulting ERASE geometry (only) and defines the minimum partition size allowed for that card. P = 1 where array partitioning at erase block boundaries is allowed (i.e. there are no special partitioning requirements).

Examples:

1. *A particular non-interleaved (DGTPL_HWIL: q = 1) PCMCIA Release 2.x Card-based memory array (DGTPL_BUS = 2) employs a new EEPROM-type of byte-wide memory device which erases in 4K bytes (DGTPL_EBS = 13) and writes in 256-byte pages (DGTPL_WBS = 9). It has no special partitioning requirements (DGTPL_PART = 1). The resulting physical geometries are:*

	Bus	x Block	x Interleave		
ERASE Geometry =	2 x	4096	x 1	=	8192 bytes
READ Geometry =	2 x	1	x 1	=	2 bytes
WRITE Geometry =	2 x	256	x 1	=	512 bytes
PARTITION Boundary =	ERASE Geometry		x 1	=	8192 bytes

2. *A particular 4-layer-interleaved (DGTPL_HWIL: q = 3) PCMCIA Release 2.x Card-based memory array (DGTPL_BUS = 2) employs a new type of word-wide flash memory device with built-in PCMCIA byte-/word-mode operation. Internal components erase in blocks that are 64KB, or 32KB times their 16-bit bus (DGTPL_EBS = 16) and write in single words (DGTPL_WBS = 1). It requires partitioning in erase block groups of four (DGTPL_PART: p = 3). The resulting physical geometries are:*

	Bus x	Block	x Interleave		
ERASE Geometry =	2 x	32768	x 4	=	262,144 bytes
READ Geometry =	2 x	1	x 4	=	8 bytes
WRITE Geometry =	2 x	1	x 4	=	8 bytes
PARTITION Boundary =	ERASE Geometry		x 4	=	1,024K bytes

5.2.8 Configuration Tuples

The Configuration and Configuration-Entry tuples describe the configurable characteristics of Memory-Only and I/O Cards. The card characteristics described by these tuples include:

- Class of interface (Memory Only, I/O or other);
- Use of Wait, Ready/Busy, Write-Protect, BVDs, and Interrupts;
- Card configurations requiring currents in excess of the nominal levels;
- Configurations of Vpp or Vcc;
- I/O port and memory mapping requirements;
- Interrupt levels;
- Unique identification of identical cards.

The Configuration tuple contains a number of fields within it which describe the interface(s) supported by the card, and, when present, the locations of the Card Configuration Registers and the Card Configuration Table.

Specific card configurations and their variations are defined in the Configuration-Entry tuples which make up the Card Configuration Table. [Note: The terms "attribute space" and "register space" are used interchangeably in this section for clarity]

5.2.8.1 CISTPL_CONFIG: Configuration Tuple

The Configuration tuple is used to describe the general characteristics of cards which can be configured for operation. This includes cards containing I/O devices or using custom interfaces. It may also describe PC Cards, including Memory-Only cards, which exceed nominal power-supply specifications, or which need descriptions of their power requirements or other information.

The default interface, in the absence of a Configuration tuple is a Memory-Only interface.

There may be no more than one Configuration tuple in the CIS.

There are presently two types of PCMCIA-defined host interfaces:

- Those supporting Memory-Card-Only interface and,
- Those supporting both the Memory- and the I/O-Card interfaces.

In addition, custom interfaces are also permitted which can be recognized and enabled in a standardized manner.

The card indicates which type(s) of custom interface(s) it can support in Custom-Interface subtuples of the Configuration Tuple. The interpretation of the rest of the bytes of the configuration tuple is determined by the size-of-fields byte, and consist of two Configuration-Registers-Descriptors, and a Reserved field.

The Card Configuration Table allows the system to determine the characteristics of the card for each allowable configuration. The system may then configure the card into any allowable configuration, or leave the card unconfigured. It may use the configuration to gracefully reject the card if the card is found to be incompatible with the system hardware and software.

Following the Reserved field there may be optional, additional tuple-formatted Interface subtuples. These relate to the interfaces available on the host. Each of these fields consists of a an Interface Subtuple Code — a link to the next subtuple and the content of the subtuple. There is presently only one such subtuple defined. However, Interface subtuple codes 80H through BFH are reserved for Vendor Specific subtuples. See Table 5-20.

Table 5-20: Configuration Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Configuration tuple code (CISTPL_CONFIG, 1AH)					
1	TPL_LINK		Link to next tuple (n-1; minimum 1)					
2	TPCC_SZ		Size of Fields Byte					
3	TPCC_LAST		Index Number of the last entry in the Card Configuration Table.					
4..	TPCC_RADR		Configuration Registers Base Address in Reg Space. 1, 2, 3 or 4 bytes depending upon the size field in TPCC_LAST					
..	TPCC_RMSK		Configuration Registers Present Mask. 1 to 16 bytes as indicated by the count in TPCC_SZ.					
..	TPCC_RSVD		Reserved area 0 - 3 bytes. Must be 0 bytes until defined.					
q+1..r	TPCC_SBTPL		The rest of the tuple is reserved for subtuples containing standardized optional additional information related to the Card Configuration.					

PCMCIA recommends, but does not require, that the Configuration Register's Base Address be placed near the beginning of the Register-Memory space.

5.2.8.1.4 TPCC_RMSK: Configuration Register Presence Mask Field for Interface Tuple

Table 5-24: Configuration Register Presence Mask Field for Interface Tuple

7	6	5	4	3	2	1	0
Configuration Register Presence Mask for Registers 7 to 0 (this byte is always present)							
Configuration Register Presence Mask for Registers 15 to 8 (this byte is present only if TPCC_RMSZ >= 1)							
...							

The presence mask for the Configuration Registers is given in this field. Each bit represents the presence (1) or absence (0) of the corresponding Configuration Register. The least-significant bit represents the lowest Configuration Register represented by the byte, and the represented registers increase with higher bit significance. Those registers which would be indicated by bytes extending beyond the end of the field are not implemented on the card.

The system uses this information together with the information in the Configuration Registers Base-Address field to determine which registers are present on the card, and where they are located in the card's register space.

The address of each configuration register is given by the formula:

*Base Address + (Register Number * 2)*

regardless of whether or not the register is present.

5.2.8.1.5 TPCC_SBTPL: Additional Information Stored in Tuple Format

The TPCC_SBTPL field extends to the end of the tuple. It contains tuple-formatted information relating to the card's configuration. These tuples are subtuples of the CISTPL_CONF tuple and cannot extend beyond the end of the configuration tuple. The chain of subtuples is ended by reaching an FFH subtuple, or reaching the end of the CISTPL_CONF tuple. Any subtuple is invalid if it contains a link field which links beyond the first byte of the tuple following the CISTPL_CONF tuple. The tuple codes of the subtuples have meaning only within the CISTPL_CONF tuple. Subtuple codes from 80H through BFH are vendor specific whereas the rest are reserved for standard definition. The only subtuples presently defined are the list-terminating subtuple, FFH (a 1-byte-long subtuple), and the CCSTPL_CIF, custom-interface subtuple.

5.2.8.1.6 CCST_CIF: Custom Interface Subtuples

The custom-interface subtuple provides a code which uniquely identifies a card-supported interface. These interfaces are referenced in the configuration-entry tuples by their relative positions in the CISTPL_CONF tuple. A 1- to 4- byte, unique, Interface Number may be followed by series of strings which describe the interface.

A maximum of four custom-interface subtuples may be present.

Vendors who wish to define custom interfaces must register with PCMCIA and will be assigned 2- or 3-byte vendor ID(s), which will provide the initial portion of the interface ID.

The first-byte codes — 41H, 81H and C1 in the field STCI_IFN— define 2-, 3- and 4- byte numbers for interfaces which may be jointly standardized by PCMCIA and JEIDA in the future, if necessary.

To simplify initial assignments, manufacturers who possess a JEDEC programmable-semiconductor manufacturer's ID may use first-byte codes 42H, 82H and C2H followed

5.2.8.1.1 TPCC_SZ: Size of Fields Byte

Table 5-21: Size of Fields Byte

7	6	5	4	3	2	1	0
TPCC_RFSZ (Reserved Size)		TPCC_RMSZ (Size of TPCC_RMSK field)				TPCC_RASZ (Size of TPCC_RADR)	

Table Field	Description
TPCC_RASZ	Indicates that the number of bytes in the Configuration Registers Base Address in Reg Space field (TPCC_RADR) of this tuple is the value of this field plus 1.
TPCC_RMSZ	Indicates that the number of bytes in the Configuration Register presence mask field (TPCC_RMSK field) of the tuple is this value plus 1.
TPCC_RFSZ	Size of area presently reserved for future use 0, 1, 2 or 3 bytes. Must be 0 at present.

5.2.8.1.2 TPCC_LAST: Card Configuration Table Last Entry Index

The Card-Configuration-Table size is a one-byte field which contains the Configuration Index Number of the last configuration described in the Card Configuration Table. Once the host encounters this configuration, when scanning for valid configurations, it will have processed all valid configurations.

Table 5-22: Card Configuration and Last Entry Index

7	6	5	4	3	2	1	0
RFU 0		Last Index					

Table Field	Description
Last Index	The Index Number of the final entry in the Card Configuration Table (the last entry encountered when scanning the CIS).
RFU 0	This area is reserved for future standardization and must be 0.

5.2.8.1.3 TPCC_RADR: Configuration Registers Base Address in REG Space

Table 5-23: Configuration Registers Base Address in REG Space

7	6	5	4	3	2	1	0
Base Address Bits 7..0 (always present)							
Base Address Bits 15..8 (present if TPCC_RASZ is >= 1)							
Base Address Bits 23..16 (present if TPCC_RASZ is >= 2)							
Base Address Bits 25..24 (present if TPCC_RASZ is = 3)							

The Base Address of the Configuration Registers, in an even byte of Attribute Memory (address of Configuration Register 0), is given in this field. The system uses this information together with the information in the Configuration Register's presence-mask field to determine which registers are present on the card and where they are located in the card's register space.

The Base Address is a field which may be from 1 to 4 bytes long depending upon the number of bits needed to represent it. High-order address bits, not present in the field, are always interpreted to be zeros. The size of this field in bytes is one greater than the value of the TPCC_RASZ size field in the TPCC_SZ byte.

by a second byte, which is their JEDEC ID. They may use up to two additional bytes of their own designation.

The subtuple is defined as follows:

Table 5-25: Custom Interface Subtuples

Byte	7	6	5	4	3	2	1	0
0	ST_CODE		Interface subtuple code (CCST_CIF, C0H)					
1	ST_LINK		Link to next tuple (m-1; minimum 1)					
2..	STCI_IFN		Interface ID Number 1 to 4 bytes Least significant byte first					
.. m	STCI_STR		Interface Description Strings					

STCI_IFN	This field provides a unique 8- to 32- bit number which identifies the interface. Within the ID number are two bits (bits 6 and 7 of byte 0) which give the size of this field. The first two or three bytes of the ID number is assigned by PCMCIA or its designer. When a vendor has been assigned a two- or three- byte initial portion the remaining one or two bytes of the field are assigned by that vendor to uniquely identify the interface.		
	Bits 7,6 of byte 0 give the Size of STCI_IFN	0	1 Byte long STCI_IFN field
		1	2 Byte long STCI_IFN field
		2	3 Byte long STCI_IFN field
		3	4 Byte long STCI_IFN field
STCI_STR	A list of strings, the first being ISO 646 IRV coded and the rest being coded as ISO alternate language strings with the initial escape character suppressed. Each string is terminated by a 0 byte and the last string, if it does not extend to the end of the subtuple, is followed by an FFH byte.		

5.2.8.2 Card Configuration Table

The initial portion of each entry of the Card Configuration Table is given in a compact, standard form. Additional information may be added to the compact information by appending tuple-formatted information (tuple-code, offset to next tuple, and tuple contents) within the interface-definition or configuration-entry tuples containing the Configuration Table Entry. Subtuple codes 80-BFH are reserved for vendor-specific configuration information.

The Card Configuration Table organization for a Memory-Only Card and an I/O Card are the same. However, a Memory-Only Card should not include I/O-specific fields or tuples. Fields related to Vcc, Vpp and timing apply to all cards.

5.2.8.3 CISTPL_CFTABLE_ENTRY: Card Configuration Table Entry Tuple

Configuration Table Entry tuples are used to specify each possible configuration of a card and to distinguish among the permitted configurations. The Configuration tuple must be located before all Configuration Table Entry tuples. The Configuration Table Entry, whose Configuration Table Index matches the last index in the Configuration tuple, must appear after all other Configuration Table Entries.

When the default bit is set in an entry's Configuration-Table index byte, that Configuration Table Entry provides all default values for following entries in the Card Configuration Table

While the default bit is not set in an entry's Configuration-Table index byte, the entry provides default values which apply only to the configuration indicated by the that entry's Configuration-Table Index (TPCE_INDX) byte.

Values which are not present in a specific entry are taken from the card's most recently scanned default entry, as indicated in the specific fields below.

Table 5-26: Configuration Table Entry Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Configuration Entry tuple code (CISTPL_CFTABLE_ENTRY, 1BH)					
1	TPL_LINK		Link to next tuple (n-1, [2 minimum])					
2	TPCE_INDX		Configuration Table Index Byte					
..	TPCE_IF		Interface-definition byte. This field is present only when the interface bit of the Configuration-Table Index Byte is set.					
..	TPCE_FS		Feature-selection field indicates which optional fields are present.					
..	TPCE_PD		Power-description structure					
..	TPCE_TD		Timing-description structure					
..	TPCE_IO		I/O-description structure					
..	TPCE_IR		Interrupt-request-description structure					
..	TPCE_MS		Memory-space- description structure					
..	TPCE_MI		Miscellaneous- information structure					
..n	TPCE_ST		Additional information about the configuration in subtuple format.					

5.2.8.3.1 TPCE_INDX: The Configuration Table Index Byte

The Configuration-Table index byte contains the value to be written to the Card Configuration Register in order to enable the configuration described in the tuple. In addition, this byte contains a bit to indicate that the configuration defined in this tuple should be taken as the default conditions for Configuration Entry tuples that follow.

Table 5-27: Configuration Index Byte

7	6	5	4	3	2	1	0
Intface	Default	Configuration Entry Number					

Configuration Entry Number	Contains the value to be written to the Card Configuration Register to enable the configuration described in the tuple	
Default	This bit indicates that this entry provides default values for succeeding entries until another entry is encountered with its Default bit set. When the default bit of the entry is set, the default conditions for following table entries are exactly those conditions specified in this entry. When the default bit of the entry is not set, the default conditions are the conditions specified by the last entry encountered with its default bit set.	
Intface See 5.2.8.3.2	0	No interface-configuration byte is present following this byte. When the default bit is set in this byte or no Configuration Entry tuple has been scanned with its default bit set then the standard, Memory- Only Interface with RDY/BSY, Write Protect and Battery Voltage Detects are present at pins but not necessarily in a Pin Replacement Register. WAIT is not generated for memory cycles. (TPCE_IF value of 70H) Otherwise, the timing is specified by the most recently scanned Configuration-Entry tuple with its default bit set.
	1	An interface configuration byte follows this byte.

5.2.8.3.2 TPCE_IF: Table Entry Interface Description Field

The Table Entry interface-description field indicates the particular card interface which the configuration uses. The interface type (Memory-Only or Memory-I/O) is specified. In addition, the card's requirements for system support of the Battery Voltage Detect, Write Protect, Ready/Busy and Wait functions may be indicated.

Table 5-28: Entry Interface Description Field

7	6	5	4	3	2	1	0
M Wait Req'd	RdyBsy Active	WP Active	BVDs Active	Interface Type			

Interface Type	0 Memory 1 I/O 2 Reserved for future standardization 3 Reserved for future standardization 4 Custom Interface 0 5 Custom Interface 1 6 Custom Interface 2 7 Custom Interface 3 8-15 Reserved for future standardization. Interfaces 4 through 7 correspond to interfaces which are defined in CCSTPL_CIF subtuples in the Configuration Tuple.The custom interface number is the relative position of the CCSTPL_CIF subtuple used by this configuration in the set of CCSTPL_CIF subtuples within the Configuration Tuple for this card.
BVDs Active	BVD1 and BVD2 Statuses are active and should be recovered from the Pin Replacement Register if not available on Pins 62 and 63.
WP Active	Write Protect Status is active and should be recovered from the Pin Replacement Register if not available on Pin 33.
RdyBsy Active	Ready/Busy Status Active and should be recovered from the Pin Replacement Register if not available on Pin 16.
M Wait Req'd	Wait Signal support required for Memory Cycles

Note: The status bits BVDs Active, WP Active, RdyBsy Active, and MWait Req'd are TRUE when set to the "1" state.

5.2.8.3.3 TPCE_INFO: Configuration Table Entry Information

This group of fields contains information which applies to this configuration. This information is in addition to, or in contrast with, the information given in the default conditions at the time the field is read.

5.2.8.3.4 TPCE_FS: Feature Selection Byte

The feature-selector byte indicates which compact descriptive fields are present within the table entry. Those fields which are present are indicated by a 1 in the corresponding bit of the selector byte. Within the entry, the fields are ordered in the order of the selector bits starting from bit 0 (LSB) to bit 7 (MSB).

Table 5-29: Feature Selection Byte

7	6	5	4	3	2	1	0
Misc	Mem Space		IRQ	IO Space	Timing	Power	

Power See 5.2.8.3.5	The power supply requirements and load characteristics for this configuration are indicated. There may be 0, 1, 2 or 3 fields following representing Vcc, Vpp, or both Vpp1 and Vpp2 in that order. The coding is as follows:	
	Code	Description
	0	No power-description structures Use the default
	1	Vcc power-description-structure Only
	2	Vcc and Vpp (Vpp1 = Vpp2) power-description-structures
	3	Vcc, Vpp1 and Vpp2 power-description-structures
Timing See 5.2.8.3.6	0	When the default bit is set in this tuple, or no Configuration-Entry tuple has been scanned with its default bit set, then no timing is specified. RDY/BSY may indicate busy indefinitely, WAIT will be active from 0 to 12 microseconds. Otherwise, the timing is specified by the most recently scanned Configuration-Entry tuple with its default bit set.
	1	A timing-description structure is present following the power-description structure.
IO Space See 5.2.8.3.7	0	When the default bit is set in this tuple, or no Configuration-Entry tuple has been scanned with its default bit set, then no I/O space is used. Otherwise, the I/O-space requirement is specified by the most recently scanned Configuration-Entry tuple with its default bit set.
	1	An I/O-space-description structure is present following the timing-description structure.
IRQ See 5.2.8.3.8	0	When the default bit is set in this tuple, or no Configuration-Entry tuple has been scanned with its default bit set, then no Interrupt is used. Otherwise, the Interrupt-request requirement is specified by the most recently scanned Configuration- Entry tuple with its default bit set.
	1	An Interrupt-request-description structure is present following the I/O- space-description structure.
MemSpace See 5.2.8.3.9	Memory address space mapping requirements for this configuration. There may be 0, 2, 4 or N bytes of information following the Interrupt Request Structure. The coding is as follows:	
	Code	Description
	0	When the default bit is set in this tuple, or no Configuration-Entry tuple has been scanned with its default bit set, then no configuration-dependent, memory-address space is used. Otherwise, the memory-address-space requirement is specified by the most recently scanned Configuration-Entry tuple with its default bit set.
	1	Single 2-byte length specified.
	2	Length (2 bytes) and Card Address (2 bytes) specified.
	3	A memory- space-selection byte followed by table-memory-space descriptors (length determined by selection byte)
Misc See 5.2.8.3.10	0	When the default bit is set in this tuple, or no Configuration-Entry tuple has been scanned with its default bit set, then the miscellaneous fields are interpreted to be all zero. Otherwise, the miscellaneous fields are specified by the most recently scanned Configuration-Entry tuple with its default bit set.
	1	A miscellaneous-fields structure is present following the memory-space-description structure.

5.2.8.3.5 TPCE_PD: Power Description Structure

Each power-description structure has the following format:

Table 5-30: Power Description Structure

Parameter Selection Byte
First Parameter Definition
...
Last Parameter Definition

Each bit in the parameter-selection byte indicates whether or not the corresponding parameter is described by parameter definitions which follow that byte. The parameter definitions are present for each logic-1 bit in the parameter-selection byte. Each byte in a parameter definition, except the last byte, has a logic-1 in bit 7 of the byte. The parameter definitions are ordered corresponding to bits 0 through 7 of the parameter-selection byte.

1) Parameter Selection Byte

Table 5-31: Power Description Structure Parameter Selection Byte

Byte	7	6	5	4	3	2	1	0
0	RFU (0)	PDwn I	Peak I	Avg I	Static I	Max V	Min V	Nom V

Nom V	Nominal operating supply voltage. In the absence of other information the nominal operating voltage has a tolerance of ±5%.
Min V	Minimum operating supply voltage.
Max V	Maximum operating supply voltage.
Static I	Continuous supply current required.
Avg I	Maximum current required averaged over 1 second.
Peak I	Maximum current required averaged over 10 milliseconds.
PDwn I	Power-down supply current required.
RFU	Reserved for future standardization.

2) Parameter Definitions

Parameter definitions consist of a first byte containing a mantissa and exponent as described below. Additional bytes appearing after the first byte serve to modify the initial value as described below. An arbitrary number of additional bytes may be defined with the last byte in the list having a 0 in bit 7. The next parameter definition, or the next field of the I/O tuple, is in the byte immediately following the last byte of the parameter definition.

Table 5-32: Power Description Structure Parameter Definitions

Byte	7	6	5	4	3	2	1	0
0	EXT	Mantissa				Exponent		
1[1]	EXT	Extension						

1. The extension bytes may be continued indefinitely until the first byte which contains a 0 in bit 7, which is the final byte of the extension

Exponent	The Exponent of the Current and Voltage Values are given below:		
	Exponent	Current Scale	Voltage Scale
	0	100 nA	10 μV
	1	1 μA	100 μV
	2	10 μA	1 mV
	3	100 μA	10 mV
	4	1 mA	100 mV
	5	10 mA	1 V
	6	100 mA	10 V
	7	1 A	100 V

Mantissa	The mantissa of the value is given by the following table:	
	0	1
	1*	1.2
	2*	1.3
	3*	1.5
	4	2
	5*	2.5
	6	3
	7*	3.5
	8	4
	9*	4.5
	AH	5
	BH*	5.5
	CH	6
	DH	7
	EH	8
	FH	9

* - These values are not permitted when the EXT bit is set.

EXT	An extension byte follows this byte.	
Extension	The extension field is defined as follows:	
	0..63H	Binary value for next two decimal digits to the right of the current value.
	64..7CH	Reserved.
	7DH	No connection (i.e. high impedance) permitted during sleep or power-down only. (Must be last extension byte).
	7EH	Zero value required (Must be only extension byte).
	7FH	No connection (i.e. high impedance) is required (Must be only extension byte)
	Extension bytes may be concatenated indefinitely. The final extension byte contains a 0 in bit 7.	

Here is an example power descriptor of a card with the following characteristics:

Operates over a Vcc range of 2.5 to 7 volts. Operating current is 1 mA standby (static), 30 mA average operating current. When in power down mode, the card requires only 75 microamps. It uses a Vpp of 12 volts ±5% at 50mA peak which need not be connected during sleep or power down.

Table 5-33: Power Description Example

Byte	Value	D7	D6	D5	D4	D3	D2	D1	D0	notes
xxx+0	02H	0	0	0	0	0	0	2 Vcc & Vpp		Feature Byte
+1	5EH	0 RFU	1 Power Down	0 No Peak I	1 Avg I	1 Static I	1 Max V	1 Min V	0 No Nominal V	Parameter Selection Byte for Vcc
+2	20H	0 No Ext	5 2.5 X				5 1 Volt			Vcc Minimum Voltage
+3	6DH	0 No Ext	Dh 7 X				5 1 Volt			Vcc Maximum Voltage
+4	04H	0 No Ext	0 1...0				4 1 mA			Vcc Static Current
+5	35H	0 No Ext	6 3.0 X				5 10 mA			Vcc Average Current
+6	EAH	1 Ext Next	Dh 7.0 X				2 10 µA			Vcc Power Down Current
+7	32H	0 No Ext	32H +.50 (gives 7.50 x 10 µA)							Vcc Power Down Current Extension Byte
+8	21H	0 RFU	0 No Pwr Down I	1 Peak I Specifie d	0 No Avg I	0 No Static I	0 No Max V	0 No Min V	1 Nominal V	Vpp Parameter Selection Byte, Nominal V and Peak I given
+9	8EH	1 Ext Next	1 1.2 X				6 10 Volts			Nominal Vpp is 12.0 Volts ±5%
+Ah	7DH	0 No Ext	7DH Vpp may be made no connect during power down and sleep modes							Extension byte indicates that No Connect is permitted on Vpp during Sleep and power Down
+BH	55H	0 No Ext	AH 5.0 X				5 10 mA			Peak Vpp current is 50 mA.

5.2.8.3.6 TPCE_TD: Configuration Timing Information

Timing information for Wait, Ready/Busy and Initialization Delay are given in scaled, extended-device-speed-code format. The first byte contains the scale factors and the other bytes contain the unscaled factors in the extended-device-speed-code format.

See Section 5.2.7.1.4 Device ID Speed Field for encoding of extended-device-speed code.

Table 5-34: Configuration Timing Information

Byte	7	6	5	4	3	2	1	0
0	Reserved Scale 7			Ready/Busy Scale			Wait Scale	

The timing descriptions, when present, occur in the same order as the fields are listed below.

WAIT Scale	This field is the power of 10 scale factor to be applied to the MAX WAIT time in extended-device-speed format which follows. The value 3 indicates that the WAIT signal is not used and the MAX WAIT Speed is not present following the timing byte.
RDY/BSY Scale	This field is the power of 10 scale factor to be applied to the MAX time in the Busy State for the Ready/Busy signal. A value of 7 indicates that Ready/Busy is not used and no maximum time is present following the timing-scale-factor byte.
Reserved Scale 7	This field is the power of 10 scale factor which is to be applied to a reserved- time definition. A value of 7 indicates that no initial or extended reserved-speed bytes follow the Ready/Busy extended-speed bytes.

5.2.8.3.7 TPCE_IO: I/O Space Addresses Required For This Configuration

The I/O-space entry consists of a set of 3- byte fields, each of which represents a range of I/O addresses occupied by the card in this mode. If more than one contiguous group of I/O addresses is present on the card, then the EXT bit is set on each group but the last.

Table 5-35: I/O Space Addresses Required For Configuration

Byte	7	6	5	4	3	2	1	0
0	Range	Bus16	Bus8	IOAddrLines				

IOAddrLines	Total number of address lines which are used by the card to determine when the card is selected.	
	0	When IOAddr Lines is zero, the card will respond to all addresses presented to the card. The system is entirely responsible for when the card is selected, and at what addresses the card is selected. The system must assign to the card a portion of the address space which is at least as large as the number of bytes indicated in the length field of the following range entry. The Base Address for the I/O space (assigned to the card by the system) must begin on a 2^N address boundary such that 2^N is greater than the number of bytes indicated in the length field.
	1-26	When I OAddrLines is non-zero, the card performs address decoding to determine when it is selected. In this case, the card and the system share the determination of when a card is actually selected. The card must indicate in IOAddrLines the highest address line (plus 1) which it decodes to determine when it has been selected. The card provides a list of ranges of addresses for which it is selected within the I/O space that it decodes. The system and the card then share the task of determining when the card is selected. The system controls when the Card Enables are activated during I/O cycles, and the card determines to which addresses it will respond when it is enabled by those Card Enables. The card returns the Input-Acknowledge signal to the system whenever the card can actually respond to an input at the I/O address on the bus.
	27-31	Reserved.

Recommended values are:

- 0 : card — responds to all I/O cycles when CE's are true, and specifies the desired address ranges to enable. It is suitable for all architectures.
- N : Where card has 2^{N-1} <I/O ports < = 2^N, and can be placed on any 2^N boundary. It is suitable for any machine architecture. For example, if a card has 32 8-bit I/O ports and the card decodes 5 address lines (A4-A0), N = 5, but the card can still be places on a 32-byte I/O boundary; it does not need to be placed on a 64-byte I/O boundary.
- 10 : 1 Kilobyte I/O address space — Software compatible for decoding in the top 768 bytes of each 1 Kbyte of I/O space. This is the same as is done in the ISA and EISA PC's.
- 16 : Full decoding of 64 Kbyte I/O Space. [Note: This is not compatible with EISA or ISA bus architectures.]

Table 5-36: 8/16 Bit Bus Width Description

Bus16	The card functions properly with 8/16-bit I/O accesses data transferred on D0 through D15.
Bus8	The card's I/O functions properly with hosts providing only 8-bit data on D0 through D7.

Table of Bus16 and Bus8 values

Bus 16/8	Card Description	Host Requirements
00	Reserved.	
01	All registers accessible only with byte accesses on D0 through D7 only.	Host must be able to access I/O on card using byte accesses only on D0 through D7. When 16-bit registers are accessed using byte accesses, both bytes must be written with the even byte first.
10	All registers accessible by 16-bit hosts which generate byte accesses for byte registers and 16-bit accesses for word registers. Eight-bit accesses to 16-bit registers are not supported. Byte accesses to odd-byte registers may take place on either D0 through D7 or D8 through D15.	Host must access 16-bit registers using 16-bit accesses, and access 8-bit registers using 8-bit accesses. IOIS16 is used to determine whether an access is to an 8- or 16-bit register.
11	All registers are accessible by both 8-bit or 16-bit. IOIs8-bit in Configuration Index Register permits host to determine whether or not to expect that all accesses to 16-bit registers will be 16-bits or 8-bits.	Host may use either method 01 or 10 above. The IOIS8-bit is used by the card to determine which access method to expect.

Range	The range of addresses to which the card responds follows this byte. If not present, the card responds to all addresses and uses IOAddrLines address lines to distinguish among its I/O ports. In this case, the amount of address space which should be allocated to the card is indicated by the number of address lines decoded by the card (e.g. 4 lines means 16 addresses).

The I/O range-descriptor byte contains the *number of I/O address ranges* field following the descriptor byte. Each I/O address range contains 0 or 1 address and 0 or 1 length. Each address has the same number of bytes as indicated by the address-size field. All length fields have the number of bytes indicated by the length-size field.

The number-of-I/O-address-ranges field contains the number of address range descriptions following minus 1. A value of 0 indicates 1 address range and a value of 15 indicates 16 address ranges.

The size fields are coded as follows:

0 — length/address is not present.
1 — length/address is 1 byte long.
2 — length/address is 2 bytes long.
3 — length/address is 4 bytes long.

The length- and size-fields may not both be zero.

Table 5-37: Length and Size fields

Byte	7	6	5	4	3	2	1	0
n	Length Size		Address Size		Number of I/O Address Ranges			

The bits which are not required to be specified should be set to 0 in both the start address and length fields.

When the address field is not present, this indicates the starting address of the block is on an arbitrary 2^N boundary, where N is the number of address lines decoded by the card.

Byte	7	6	5	4	3	2	1	0
0..	Start of I/O Address Block 0, 1, 2 or 4 bytes long. Address bits in bytes which are not present are zeros.							
0-4..	Length of I/O Address Block -1 0, 1, 2 or 4 bytes long. Length bits in bytes which are not present are zeros.							

5.2.8.3.8 TPCE_IR: Interrupt Request Description Structure

When the IRQ bit is set, it indicates that an interrupt description follows the I/O address bytes, if any. The interrupt request levels specified by the Configuration Table Entry Tuple describe the preferred routing for the card's -IREQ line. Routing of the -IREQ interrupt is performed by the host system which actually determines the system interrupt level used. A client which is the sole consumer of the card's -IREQ interrupt may elect to route the -IREQ line to a level not specified by the Configuration Table Entry Tuple. A generic card configuration utility which is not the ultimate consumer of the card's -IREQ interrupt should only route to the specified interrupt levels. There are either one or three interrupt description bytes as shown below:

Table 5-38: TPCE_IR Interrupt Request Description Structure

Byte	7	6	5	4	3	2	1	0
0	Share	Pulse	Level	Mask	IRQN Line 0-15			
					VEND	BERR	IOCK	NMI
1*	IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0
2*	IRQ15	IRQ14	IRQ13	IRQ12	IRQ11	IRQ10	IRQ9	IRQ8

* These bytes present only when bit 4 of byte 0 is one. See Mask below.

Share	The card contains interrupt sharing support logic in the Card Control and Status Register which may be used to allow interrupt levels to be shared among several cards.
Pulse	When this bit is one, the interrupt request pin in this mode may be pulsed low momentarily to request an interrupt.
Level	The Level Bit, when set, indicates that in this configuration the card's interrupt request pin can remain asserted at the active low- level until the interrupt is serviced. When a configuration is capable of supporting level-mode interrupts, the IRQLEV bit in the Configuration Register controls whether the level- or pulsed-mode interrupts are selected.
Mask	The Mask bit indicates whether the possible destinations of the interrupt request line are indicated by a single interrupt line value, or a mask of possible interrupt lines. When this bit is zero, the destination is given by the IRQN field, formed from right-hand nibble of byte 0, which selects 1- of-16 possible interrupt request lines. In this case, bytes 1 and 2 are not present. When this bit is a one, the 4 least-significant bits of byte 0 are masks for possible interrupt destinations. In addition, two additional bytes are present which contain mask bits for 16 possible interrupt request lines.
NMI	
IOCK	
BERR	
VEND	These bits are valid only when the Mask bit is 1. When set, each of these bits indicates that the corresponding special- interrupt signal may be assigned by the host to the card's interrupt-request pin. Systems are not required to support these interrupts. NMI is non-maskable interrupt. IOCK is the I/O-check signal. BERR is Bus-Error signal. VEND is a Vendor-Specific signal.
IRQ0-15	These bytes are present only when the Mask bit is a 1. When set, each of these bits indicates that the corresponding interrupt signal may be assigned by the host to the card's interrupt-request pin. IOCK is the I/O-check signal. General purpose systems must support all those interrupts which are available on the I/O bus normally associated with the environment they are supporting.

5.2.8.3.9 TPCE_MS: Memory Space Description Structure

The MemSpace description indicates the card requires a configuration-entry-related portion of its Common-Memory space be mapped into the host's address space. This memory may be either direct mapped (no address translation is performed by the host), or may require that the host translate the host address to the corresponding card address in order to support the card. In either case, for the host to support the card under this configuration, the host must access the card when the corresponding addresses are accessed by software.

The MemSpace field is encoded as follows:

0 — No memory mapping required.

1 — Single length specified. Card's Base Address is 0 and any host address may be mapped. The length is 2-bytes long and is the number of 256-byte pages which are to be mapped.

2 — Single length and card Base Address specified. Host address equals card address. The length and card address are each 2-bytes long and represent the number of 256- byte pages which are mapped, and the starting page, which is mapped, respectively. The length field appears before the card address field.

3 — A Memory-Space descriptor byte is present, followed by a table of Memory-Space descriptors.

Table 5-39: Memory Space Descriptor Byte

Byte	7	6	5	4	3	2	1	0
0*	Host Addr	Card Address Size		Length Size		Number of Windows		

* This byte is present only if the MemSpace field in the feature Selection byte is 3.

The memory descriptor contains the following fields:

Number of Windows	The number of window descriptors following minus 1.
Length Size	The size of the length fields in bytes. The length is in 256-byte pages
Card Address Size	The size of the card address fields in bytes. The card address is in 256-byte pages.
Host Address	This bit is set if a host address field, the same size as the card-address field, is present. This bit is zero, and no host-address field is present, when the host address is arbitrary.

Bytes	7	6	5	4	3	2	1	0
0-3	Length of Window in 256-byte blocks. 0 - 3 bytes depending upon length-size field above.							
0-3	Card Address for Window in 256 byte-blocks. 0 - 3 bytes depending upon card address-size field above							
0-3	Host Address for Window in 256-byte blocks.Present only if host-address bit is set to 1 above. 0 - 3 bytes depending upon card address-size field above							

Length	The length of the block is given in units of 256 bytes. The least significant part appears in byte 0.
Card Address	This field contains the address to be accessed on the card corresponding to the host address. The least significant part appears in byte 0.
Host Address	The physical address in the host-address space where the block of memory must be placed. This field is present only when the host-address bit is set to 1. The least significant part appears in byte 0.

5.2.8.3.10 TPCE_MI: Miscellaneous Features Field

When the Misc bit is set in the feature-selection byte, it indicates that the byte(s) following the memory-address-space bytes contain additional information. There is presently one byte defined in the Misc structure, however, additional bytes may be defined in the future.

Table 5-40: Miscellaneous Features Field

Byte	7	6	5	4	3	2	1	0
0	EXT	Resv'd (0)	Pwr Down	Read Only	Audio	Max Twin Cards		

Max Twins	This field is used to indicate the card requires that identical cards installed in the system be differentiated from each other in a sequential manner. For example, first twin is card 0, second is card 1, and so on. This allows the cards to share I/O ports and interrupts in a manner consistent with some peripherals commonly used in PC computers, such as fixed-disk drives. The max-twins field specifies the maximum number of other identical cards which can be configured identically to this card. This permits more than one card to be installed in host which responds to the identical I/O addresses. The host allows the cards to distinguish among themselves by writing their "Copy" numbers (e.g. 0, for the first card, 1 for the second, etc.) into the copy field of the Socket and Copy Register in the Card Configuration Registers.
Audio	This bit indicates the card allows the BVD2 signal to be used as Audio Waveform for the speaker. *This operation is controlled by the Audio Enable Bit in the Card Control and Status Configuration Register.*
Read Only	This bit indicates the card contains a data-storage medium which is read-only for this configuration. There may be other configurations for which the storage medium is read/write.
Pwr Down	This bit indicates the card supports a power-down mode controlled by the power-down bit in the Control and Status Register.
Resv'd (0)	These bits are reserved for future definition and must be 0.
EXT	An extension follows this byte. A series of extension bytes, which will be defined by PCMCIA, will be terminated when an extension byte is encountered which does not have the EXT bit set.

5.2.8.4 STCE_EV: Environment Descriptor Subtuple

The environment-descriptor subtuple is an optional subtuple used to describe the environment in which the configuration is used. It describes the system and, optionally, the operating system for which the configuration is intended.

The subtuple contains one or more strings, the first of which contains 3 parts. The three parts are: 1) System Name, 2) Operating System Name, and 3) Comment.

The Operating System Name is preceded by a colon (:) and the comment is preceded by a semicolon (;).

The remaining strings contain comment (and, optionally, the system name and OS name) in alternate languages.

Each string is terminated by a 00H byte. The list of strings is terminated by reaching the end of the subtuple

When the default bit is set in this configuration-entry tuple, the environment descriptor becomes the default-environment descriptor for succeeding entries as well.

Table 5-41: Environment Descriptor Subtuple

Byte	7	6	5	4	3	2	1	0
0	ST_CODE		Code value indicating that this is the environment subtuple (STCE_EV, C0H).					
1	ST_LINK		Link to next subtuple (at least m-1).					
2..m	STEV_STRS		A list of strings, the first being ISO 646 IRV coded, and the rest being coded as ISO alternate language strings, with the initial escape character suppressed. Each string is terminated by a 0 byte, and the last string, if it does not extend to the end of the subtuple, is followed by an FFH byte.					

5.2.8.5 STCE_PD: Physical Device Name Subtuple

The physical-device-name subtuple is an optional subtuple used to describe the physical device being implemented by the configuration.

The subtuple contains one or more strings, the first of which contains up to two parts. The first part is the physical-device name; the second part, preceded by a semicolon (;) is comment.

The remaining strings contain comment (and, optionally, the physical-device name) in alternate languages.

When the default bit is set in this configuration-entry tuple, the environment descriptor becomes the default-environment descriptor for succeeding entries as well.

Table 5-42: Physical Device Name Subtuple

Byte	7	6	5	4	3	2	1	0
0	ST_CODE		Code value indicating that this is the environment subtuple (STCE_PD, C1$_H$).					
1	ST_LINK		Link to next subtuple (at least m-1).					
2..m	STPD_STRS		A list of strings, the first being ISO 646 IRV coded, and the rest being coded as ISO alternate language strings, with the initial escape character suppressed. Each string is terminated by a 0 byte, and the last string, if it does not extend to the end of the subtuple, is followed by an FF$_H$ byte.					

5.2.8.6 Additional I/O Feature Definitions within Entry

Additional information about the configuration may be embedded, at the end of, but within, a table entry in the Card Configuration Table. The information takes the form of configuration-entry-specific subtuples. These tuples begin at the byte following the last feature-description byte indicated by the feature-selector byte. These additional tuples must be in standard tuple form (tuple code, followed by offset to next tuple, followed by tuple info) and may not extend beyond the size for the entry, indicated by the offset to the next table entry. The tuple-codes definitions used are valid only within the interface- definition tuple, or the configuration-entry tuple.

5.2.9 Use of Additional Tuples

Cards that comply only with the standard's basic-compatibility layer need only contain the basic-device-information tuple for the Common Memory space. Cards that comply with one of the data-format standards (listed in Section 5.3.3) will need to supply additional information as specified in that section. In many cases, an implementation will need only to verify that certain subfields in specific tuples are compatible with its requirements.

5.3 Data Recording Formats (Layer 2)

This level defines the data-recording format for the card. If none of the layer-2 headers are present, software should assume that the card is organized as an unchecked sequence of bytes.

Card data-recording formats fall into two categories:

- Disk-like — the card consists of a number of data blocks, where each block consists of a fixed number of bytes. These blocks correspond to the sectors of rotating disk drives. Conceptually, an entire block must be updated if any byte in the block is to be changed.
- Memory-like — the card is treated as a sequence of directly-addressable bytes of data. Formats are further categorized according to how error checking is performed.

This standard recognizes four basic possibilities:

- Unchecked — no data checking is performed at the data-format layer.
- Checked with in-line codes — data checking is performed by the data-format layer using check codes. The check code for a given block is recorded immediately after the block.
- Checked with out-of-line codes — data checking is performed by the format layer using check codes. The check code for a given block is recorded in a special table that resides separately from the data blocks.
- Checked over entire partition — data checking is performed only over the complete partition.

This standard recognizes two kinds of check codes: arithmetic checksum and CRC. Arithmetic checksums are typically one- or two-bytes long; CRCs are always two-bytes long.

When cards with 16-bit data paths are used to record byte data, it is necessary to specify how the bytes of the data card correspond to sequential bytes of data. In this standard, all disk-like organizations require that bytes be assigned to words, with the lowest-byte-address mapping to the least-significant byte of the word, and subsequent-byte-address mapping to increasingly significant bytes.

Memory-like formats also require that the byte mapping be specified. For maximum flexibility, both little-endian and big-endian byte orders are supported.

5.3.1 Card Information Tuples

The tuples listed in the following subsections provide generic information about how the card is to be used.

5.3.1.1 CISTPL_VERS_2: The Level-2 Version and Information Tuple

The Level-2 information tuple serves to introduce information pertaining to the logical organization of the card's data. The layout of the Level-2 tuple is shown in Table 5-43. This tuple should appear only one time in a CIS.

Table 5-43: Level-2 Information Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Code value indicating that this is the Level-2 tuple (CISTPL_VERS_2, 40_H).					
1	TPL_LINK		Link to next tuple (at least m-1).					
2	TPLLV2_VERS		Structure version (00_H).					
3	TPLLV2_COMPLY		Level of compliance claimed.					
4 .. 5	TPLLV2_DINDEX		Byte address of first data byte in card (LSB first).					
6 .. 7	TPLLV2_RSV6, TPLLV2_RSV7		Reserved; must be zero.					
8 .. 9	TPLLV2_VSPEC8, TPLLV2_VSPEC9		Vendor-specific bytes.					
10	TPLLV2_NHDR		Number of copies of CIS present on the device.					
11 .. k	TPLLV2_OEM		Vendor of software that formatted card (ASCII, variable length, terminated with a NUL[00_H]).					
k+1 .. m	TPLLV2_INFO		Informational message about the card (ASCII, variable length, terminated with a NUL [00_H]).					

TPLLV2_VERS represents the standardization version of the tuple. This byte should always be zero.

TPLLV2_COMPLY indicates the claimed degree of compliance with this standard. At present, this should always be zero.

TPLLV2_DINDEX specifies the address of the card's first data byte. Setting this to non-zero reserves bytes at the beginning of Common Memory. [Note: The first data byte on the card must always be somewhere in the first 64 Kbytes of the card. This field should be consistent with information provided in the format tuple (if that tuple is present)].

TPLLV2_NHDR specifies the number of copies of the CIS that are present on the card. For compatibility with this standard, this value should be 1.

TPLLV2_OEM specifies the vendor of the machine, or format program, that formatted the card. This is a text string terminated by a NUL byte (00H). The value of TPLLV2_OEM, combined with the value of TPLLV2_INFO, determines how to interpret vendor-specific fields in the Level-2 tuples.[1] For alternate languages, CISTPL_ALTSTR tuples may follow this tuple, specifying the string value to be substituted when using alternate languages.

TPLLV2_INFO contains a text message terminated by a NUL byte (00H). This message should be displayed to users, by a computer, whenever the host needs to describe the type of card that is in the drive. For alternate languages, CISTPL_ALTSTR tuples may follow this tuple, specifying the string value to be substituted when using alternate languages.

[Note: If the computer system's format-routine determines that the card is already formatted, it will display a message like: *Caution! This card contains data for <info>, from <vendor>*. The contents of the information field should be chosen appropriately. For example, a VCR setup card for a VCR by AlphaBeta Electronics might have <info> as "Model 9770 VCR" and the <vendor> field would be "AlphaBeta".]

The characters used in TPLLV2_INFO and TPLLV2_OEM shall be chosen from the printing 7-bit ISO 646 IRV set (codes 20H through 7EH, inclusive).

TPLLV2_RSV6 and TPLLV2_RSV7 are reserved for use in future versions of this standard. They shall be set to zero.

TPLLV2_VSPEC8 and TPLLV2_VSPEC9 are vendor specific. If not used, they shall be set to zero.

5.3.1.2 CISTPL_DATE: The Card Initialization Date Tuple

This optional tuple indicates the date and time at which the card was formatted.

Only one CISTPL_ DATE tuple is allowed per card.

Its format is given in Table 5-44.

Table 5-44: Card Initialization Date Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Initialization-date tuple code (CISTPL_DATE, 44H).					
1	TPL_LINK		Link to next tuple (at least 4).					
2	TPLDATE_TIME: MMM_{lo}			SSS				
3	HHH					MMM_{hi}		
4	TPLDATE_DAY: MON_{lo}			DAY				
5	YYYY							MON_{hi}

[1] The PCMCIA will maintain a registry of vendor names.

Bytes 2-3 (TPLDATE_TIME) indicate the time at which the card was initialized. It should be a 16-bit number stored with LSB first.

- The field HHH contains the hour at which the card was initialized. It is a number between 0 and 23.
- The field MMM contains the minute at which the card was initialized. It is a number between 0 and 59.
- The field SSS represents the two-second interval at which the card was initialized. It is a binary number between 0 and 29. To convert SSS to seconds, it should be multiplied by two.

Bytes 4-5 (TPLDATE_DAY) indicate the date the card was initialized. It should be a 16-bit number stored with LSB first.

- The field YYYY represents the year. It is a binary number between 0 and 127, with 0 representing the year 1980.
- The field MON represents the month. It is a binary number between 1 and 12, with 1 representing January.
- The field DAY represents the day. It is a binary number between 1 and 31.

If the date and time components of the date are both zero, this should indicate that the date and time when the card was first initialized were unknown.

5.3.1.3 CISTPL_BATTERY: The Battery-Replacement Date Tuple

This optional tuple shall be present only in cards with battery-backed storage. It indicates the date at which the battery was replaced, and the date at which the battery is expected to need replacement.

Only one CISTPL_ BATTERY tuple is allowed per card.

Its format is given in Table 5-45.

Table 5-45: Battery Replacement Date Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Initialization-date tuple code (CISTPL_BATTERY, 45_H).					
1	TPL_LINK		Link to next tuple (at least 4).					
2	TPLBATT_RDAY: MON$_{lo}$			DAY				
3	YYYY					MON$_{hi}$		
4	TPLBATT_XDAY: MON$_{lo}$			DAY				
5	YYYY							MON$_{hi}$

Bytes 2-3 (TPLBATT_RDAY) indicate the date on which the battery was last replaced. It should be a 16-bit number stored with LSB first. This field has the same interpretation as the field TPLDATE_DAY.

Bytes 4-5 (TPLBATT_XDAY, "expiration day") indicate the date on which the battery should be replaced. This field has the same format as TPLBATT_RDAY.

If either field is zero, it indicates that the corresponding date was not known when the tuple was recorded.

5.3.2 Data Recording Format Tuples

All information about a card's data-recording format is given in special tuples in the Card Information Structure. Each card that conforms to this standard's Layer 2 shall contain at least one format tuple defining how the data is recorded on the card.

If the format is disk-like, the format tuple may be followed by a geometry tuple. This tuple indicates the cylinder, track and sector layout for operating environments that need to treat all mass-storage devices in that way.

If the format is memory-like, the format tuple may be followed by a byte-order tuple. The byte-order tuple specifies two independent (but related) parameters:

- How multi-byte numbers are recorded on the media, and
- How byte addresses are assigned within each word (for cards with 16-bit or wider data-paths).

5.3.2.1 CISTPL_FORMAT: The Format Tuple

The format tuple defines the data recording format for a region (usually all) of a card. Its layout is shown in Table 5-46.

Table 5-46: Format Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Format tuple code (CISTPL_FORMAT, 41H).					
1	TPL_LINK		Link to next tuple (n-1; at least 12, typically 20)					
2	TPLFMT_TYPE		Format type code (TPLFMTTYPE_xxx); see Table 5-47.					
3	TPLFMT_EDC							
	RFU	Error-detection-code type.				EDC Length		
4 .. 7	TPLFMT_OFFSET		Byte address of the first data byte in this partition.					
8 .. 11	TPLFMT_NBYTES		Number of data bytes in this partition.					
12 .. n	Additional information, interpreted based on value of TPLFMT_TYPE.							

Each format tuple implicitly begins a partition tuple set. Subsequent geometry, byte order, *software interleave* and data organization tuples are implicitly associated with the preceding format tuple.

Byte one of the tuple specifies the link to the next tuple, and, therefore, (implicitly) the length of this tuple. Two ranges of values are permitted. Normally, the value will be at least 20 (014H), however, if the format tuple is specifying a memory-like format, the value may be as little as 12 (0CH), as bytes 13 through 21 must be zero for memory-like formats. If the partition does not use error-detecting codes, then the TPLFMT_EDCLOC field may be omitted.

Byte two of the tuple (TPLFMT_TYPE) specifies the kind of format used for this partition. The permitted values for this field are given in Table 5-47.

Table 5-47: Format Type Codes

Code	Name	Description
0	TPLFMTTYPE_DISK	This partition uses a disk-like format.
1	TPLFMTTYPE_MEM	This partition uses a memory-like format.
2 .. 7FH		(Reserved for future standardization.)
80h .. FFH	TPLFMTTYPE_VS	This partition uses a vendor-specific format.

- Byte 3 (TPLFMT_EDC) specifies the error-detection method, and the length of the error-detection code. Byte 3 is generally only meaningful for disk-like formats. Bit 7 is reserved; it must be zero. Bits 3-6 specify the error-detection code. The legal values are given in Table 5-48. Bits 0-2 (TPLFMT_EDCLEN) specify the length in bytes of the error-detection code; this is a number between 0 and 7. The legal values for the length field are determined by the error-detection method in use.
- Memory-like regions may use the PCC method of error detection.

Table 5-48: Error Detection Type Codes

Code	Name	Description
0	TPLFMTEDC_NONE	No error-detection code is used. If the length field is non-zero, space will be reserved for the check code, but no checking will be performed.
1	TPLFMTEDC_CKSUM	An arithmetic checksum is used to check the data. The length field must be 1 if this code is selected. See Section 5.3.2.1.3 for details in calculating the checksum.
2	TPLFMTEDC_CRC	A cyclical redundancy check is used to check the data. The length field must be 2 if this code is selected. The CRC value is always recorded low-order byte first; see Section 5.3.2.1.4 for details.
3	TPLFMTEDC_PCC	An arithmetic checksum is used to check the data. However, a single checksum is provided for the entire data partition. This technique is intended for use with static data on ROM or OTPROM cards. The PCC code itself is recorded in byte 18 of the tuple (field TPLFMT_EDCLOC, byte 0). The length field must be 1 if this option is selected.
4 .. 7H		(Reserved for future standardization.)
8H .. FH	TPLFMTEDC_VS	A vendor-specific method of error checking is used.

The code in TPLFMT_EDC only specifies the method to be used to verify data integrity. The value in TPLFMT_EDCLOC must be consulted to determine whether the code is to be interleaved with the data, or stored in a separate table.

Bytes 4-7 (TPLFMT_OFFSET) specify the absolute byte address of the first data byte governed by this tuple. The value is stored as a 32-bit quantity with LSB first.

Bytes 8-11 (TPLFMT_NBYTES) specify the number of bytes in the partition, including (if present) the error-detection codes. The value is stored as a 32-bit quantity with LSB first.

5.3.2.1.1 The Format Tuple for Disk-like Regions

When the TPLFMT_TYPE field of the format tuple has the value TPLFMT_DISK, bytes 12 through 21 of the tuple are interpreted as shown in Table 5-49.

Table 5-49: Format Tuple for Disk-like Regions

Byte	7	6	5	4	3	2	1	0
12 .. 13	TPLFMT_BKSZ		Block size. For unblocked formats, this value should be 0. This field corresponds to the number of data bytes per sector. The value in this field must be a power of 2.					
14 .. 17	TPLFMT_NBLOCKS		Number of data blocks in this partition.					
18 .. 21	TPLFMT_EDCLOC		Location of the error-detection code. If zero, the error-detection code is interleaved with the data blocks. If non-zero, the error-detection code is stored in a linear table starting at the specified address on the card. If using PCC, the first byte of this field contains the check code and bytes 19 .. 21, if present, must be zero.					

Bytes 12-13 (TPLFMT_BKSZ) specify the number of data bytes in each block in the partition. This value does not include error-check bytes. The value in this field must be a power of 2 between 128 and 2048. This standard recommends that 512 be used wherever possible. The value is stored as a 16-bit quantity with LSB first.

Bytes 14-17 (TPLFMT_NBLOCKS) specify the number of data blocks in the partition. This value is stored as a 32-bit quantity with LSB first. The quantity:

TPLFMT_NBLOCKS *(TPLFMT_BKSZ + TPLFMT_EDCLEN)

shall be less than or equal to TPLFMT_NBYTES.

Bytes 18-21 (TPLFMT_EDCLOC) specify where the error-detection codes are stored. This value is stored as a 32-bit quantity with LSB first. If the value stored in this location is zero, or if this field is not present, then codes (if present) are interleaved with the data blocks, with the code for a given data block to follow immediately after that block. If the value stored in this location is non-zero, it shall be the address of the first byte of the error-detection code table. This table shall be an array of values, with TPLFMT_NBLOCKS entries, containing the error-detection codes for each data block. Each entry in the table shall have a byte length as indicated by TPLFMT_EDCLEN. The value stored in TPLFMT_EDCLOC shall be at least the value in TPLFMT_OFFSET, and shall be no greater than the result of TPLFMT_OFFSET + TPLFMT_NBYTES - (TPLFMT_EDCLEN · TPLFMT_NBLOCKS).[1]

If PCC error checking is selected, then the TPLFMT_EDCLOC field is used to add the actual CRC value, rather than pointing to the cell that holds the PCC.

The bit TPLFMT_EDC_RFU is reserved for future use and shall always be zero.

[1] In other words, the table must be entirely contained in the range of bytes between TPLFMT_OFFSET and TPLFMT_OFFSET + TPLFMT_NBYTES - 1. Since the first data byte of block 0 resides at TPLFMT_OFFSET, the standard requires that the EDC table appear after all the data blocks in the partition. The standard does not require that the table occur immediately after the last block, nor does it preclude use of spare space for vendor-specific purposes.

Table 5-50 summarizes some possible error-detection strategies.

Table 5-50: Error Detection Format Summary

EDC Format	EDC Length	EDC Location	Description
TPLFMTEDC_NONE	0	0	No error checking is performed and no room is reserved for error-detection tables. The data blocks are recorded sequentially.
TPLFMTEDC_NONE	2	0	No error checking is performed, but room is reserved for a two-byte error-detection code after each data block.
TPLFMTEDC_NONE	1	non-zero	No error checking is performed, but room is reserved for an out-of-line table of error-detection codes, with one byte per data block. The data blocks themselves are recorded contiguously.
TPLFMTEDC_CKSUM	1	non-zero	Data is checked using a one-byte arithmetic checksum of the data. The checksum is stored in an out-of-line table. The data blocks themselves are recorded contiguously.
TPLFMTEDC_CRC	2	0	Data is checked using SDLC CRC codes. The check-code for a data block is stored immediately following the data block.
TPLFMTEDC_PCC	1	special	Entire partition is checked using a one byte arithmetic checksum. The checksum is stored in the TPLFMT_EDCLOC field of the tuple itself.

5.3.2.1.2 The Format Tuple for Memory-like Regions

When the TPLFMT_TYPE field of the format tuple has the value TPLFMT_MEM, bytes 12 through 21 of the tuple are interpreted as shown in Table 5-51.

Table 5-51: Format Tuple for Memory-like Regions

Byte	7	6	5	4	3	2	1	0
12	TPLFMT_FLAGS		Various flags				AUTO	ADDR
	(Reserved)							
13			(Reserved; must be zero)					
14 .. 17	TPLFMT_ADDRESS		Physical address at which this memory partition should be mapped, if so indicated by TPLFMT_FLAG. Four bytes stored with LSB first.					
18 .. 21	TPLFMT_EDCLOC		Error-detection code location, with same meaning as for disk-like regions. Used for PCC checking only. Byte 18 holds the check value, and bytes 19 .. 21 must be zero or omitted.					

As with the format tuple for disk-like regions, zero bytes may be omitted at the end of the format tuple for memory-like regions. To do this, the next tuple must begin immediately after the last non-zero byte in the format tuple.

Byte 12 (TPLFMT_FLAGS) contains several control bits.

- Bit 0 (TPLFMTFLAGS_ADDR), if set, indicates that bytes 14-17 (TPLFMT_ADDRESS) represent a physical address to be associated with the first byte of this region. If clear, bytes 14-17 do not represent a physical address and must be zero.
- Bit 1 (TPLFMTFLAGS_AUTO) tells the system whether to automatically map the region into memory when the card is inserted (or at system start-up). If set, the system should attempt to map the region into memory. If TPLFMTFLAGS_ADDR is set, the system should attempt to map the code at the specified address. A system shall ignore these fields if it cannot perform the specified mapping. It may also, at the designer's option, ignore these fields even if it could perform the mapping.

Byte 13 is reserved and, if present, must be set to zero.

Bytes 14 through 17 (TPLFMT_ADDRESS) represent the physical address at which the partition should be mapped into the host's address space. For 80x86-family machines, this is a linear address, not a segment/offset address. If the flag TPLFMTFLAGS_ADDR is not set, then this field is reserved and must be zero.

[Note: A system can comply fully with this standard and not honor these fields. The automatic mapping feature is not intended for general-purpose use, or for building interchangeable BIOS extensions for general-purpose systems. The standard's automatic mapping feature is included for use in low-cost embedded systems, and is not intended as a general, execute-in-place, specification. Many important issues are deliberately not addressed by this part of the standard, such as what to do when the card is removed, or how to resolve conflicts when cards in different sockets both need to be mapped to a specific physical address. It is anticipated that general purpose DOS-based systems will ignore these fields.]

Bytes 18-21 (TPLFMT_EDCLOC) have the same meaning for memory-like regions that they have for disk-like regions. A memory-like region has only two options for error checking: none or PCC. Therefore, if this field is used, byte 18 contains the check code, and bytes 19-21 are reserved and must be zero.

5.3.2.1.3 Arithmetic Checksums As Error-Detection Codes

Arithmetic checksums shall be computed by summing together all the data bytes of the block using eight-bit, twos-complement addition and ignoring any overflow that occurs. The resulting sum shall be stored in an external table (for block-by-block checksum,) or in the format tuple itself (for PCC checking).[1]

5.3.2.1.4 CRC Error-Detection Codes

CRC codes shall be computed using the SDLC algorithm.[2] In this algorithm, the data to be checked is considered as a serial-bit stream, with the low-order bit of the first byte taken as the first bit of the stream. This bit stream is conceptually taken as the coefficients of a polynomial in x^n, where n is the number of bits in the stream, and where the first bit is the coefficient of the term in x^{n-1}. This polynomial is multiplied by x^{16} and then divided (modulo 2) by the polynomial $x^{16} + x^{12} + x^5 + 1$, leaving a remainder of order 15 or less.[3] The one's complement of this remainder is the error-check code. It is recorded with the complemented coefficient of x^{15} as its least-significant bit, and with the complemented coefficient of x^0 as its most-significant bit.

The SDLC CRC has a convenient property. When the check code is appended to the data stream, and the algorithm is run on the result, the remainder will always be $x^{12} + x^{11} + x^{10} + x^8 + x^3 + x^2 + x^1 + x^0$ (assuming that neither the data nor the CRC have been corrupted).

Despite its complicated formal definition, the SDLC CRC is quite easy to compute both in hardware and in software. Commercially available chips (such as the 9401) can compute the CRC directly from a serial-data stream. There are several well-known methods for computing the CRC, one-byte-at-a-time, using a lookup table. Even so, computing a CRC in software is somewhat slower than computing a simple checksum.

[1] This method has the disadvantage that the checksum of a block of zero data is also zero; however, it is consistent with current practice. Most implementors will not want to interleave checksums and data. If some do, the standard should introduce another error-detection code type, one's complement of checksum.

[2] Also known as CRC-CCITT or HDLC CRC.

[3] If the register initial-condition is set to all ones, the CRC code for a block of all zeros will be non-zero.

5.3.2.1.5 Byte Mapping for Disk-Like Media

Within a disk-like partition, cards with 16-bit data paths shall be byte mapped with the lowest-byte address of each word corresponding to the least-significant byte of that word, and with increasing-byte addresses corresponding to increasingly significant bytes.

5.3.2.2 CISTPL_GEOMETRY: The Geometry Tuple

This tuple shall only appear in partition descriptors for disk-like partitions. It provides instructions to those operating systems that require that all mass-storage devices be divided into cylinders, tracks and sectors.

Table 5-52: Geometry Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Geometry tuple code (CISTPL_GEOMETRY, 42H).					
1	TPL_LINK		Link to next tuple. (at least 4)					
2	TPLGEO_SPT		Sectors per track.					
3	TPLGEO_TPC		Tracks per cylinder.					
4 .. 5	TPLGEO_NCYL		Number of cylinders, total.					

Byte 2 (TPLGEO_SPT) specifies the number of sectors per simulated track on the memory card. This is a number between 1 and 255. A value of zero is not permitted.

Byte 3 (TPLGEO_TPC) specifies the number of tracks per simulated cylinder on the device. This is a number between 1 and 255. A value of zero is not permitted.

Bytes 4-5 (TPLGEO_NCYL) specify the number of simulated cylinders on the device. This is a number between 1 and 65535, stored as a 16-bit integer with LSB first.[1]

The product

*TPLGEO_NCYL * TPLGEO_TPC * TPLGEO_SPT*

shall be less than or equal to the number of blocks recorded in field TPLFMT_NBLOCKS of the format tuple (subsection 5.3.2.1.2).

5.3.2.3 CISTPL_BYTEORDER: The Byte-Order Tuple

This tuple shall only appear in a partition tuple set for a memory-like partition. It specifies two parameters: the order for multi-byte data, and the order in which bytes map into words for 16-bit cards.

Table 5-53: Byte Order Tuple

Byte	7	6	5	4	3	2	1	0
0	TPL_CODE		Tuple code (CISTPL_BYTEORDER, 43).					
1	TPL_LINK		Link to next tuple; should be at least 2.					
2	TPLBYTE_ORDER		Byte order code: see Table 5-54.					
3	TPLBYTE_MAP		Byte mapping code: see Table 5-55.					

[1] This value is one greater than the same quantity as represented by the PC BIOS. This standard records the number of simulated cylinders; the PC BIOS records the maximum cylinder number. Since cylinder numbers on the PC start at zero, the maximum cylinder number on the PC is one less than the number of cylinders.

Byte 2 (TPLBYTE_ORDER) specifies the byte order for multi-byte numeric data. Symbolic codes for this field begin with the text "TPLBYTEORD_", and are listed in Table 5-54.

Table 5-54: Byte Order Codes

Code	Name	Description
0	TPLBYTEORD_LOW	Specifies that multi-byte numeric data is recorded in little-endian order.
1	TPLBYTEORD_HIGH	Specifies that multi-byte numeric data is recorded in big-endian order.
2-7FH		Reserved for future standardization.
80H-FFH	TPLBYTEORD_VS	Vendor-specific.

Byte 3 (TPLBYTE_MAP) specifies the byte mapping for 16-bit or wider cards. Symbolic codes for this field begin with the text "TPLBYTEMAP_", and are listed in Table 5-55.

Table 5-55: Byte Mapping Codes

Code	Name	Description
0	TPLBYTEMAP_LOW	Specifies that byte 0 of a word is the least-significant byte (multi-byte cards).
1	TPLBYTEMAP_HIGH	Specifies that byte 0 of a word is the most-significant byte (multi-byte cards).
2-7FH	Reserved for future standardization.	
80H-FFH	TPLBYTEMAP_VS	Vendor-specific.

If a byte-order tuple is not present, the data shall be recorded using little-endian byte order (TPLBYTEORD_LOW), and shall be mapped with byte 0 of each word corresponding to the least-significant byte (TPLBYTEMAP_LOW).

For applications involving DOS file systems, little-endian byte order and low-to-high byte mapping are mandatory.

5.3.2.4 *Software Interleave Tuple*

This tuple allows the software interleaving of data within a partition on a card. The presense of this tuple depicts that the partition that it is associated with is 2n (n=TPL_SWIL_INTRLV) interleaved. The presence of this tuple implies the non-sequential physical address sequence used to read and write data. The data is reconstructed after reading the data from the card using an n-way interleaved sequence. Note, this tuple is associated with the previous format tuple.

Byte	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	*TPL_CODE*		*Tuple code for this tuple.* *(CISTPL_SWIL, 23h)*					
1	*TPL_LINK*		*Link to next tuple (at least 2).*					
2	*TPL_SWIL_INTRLV*		*Value = n, where data is recorded via software interleaving by a factor of 2n. The partition total size divided by this interleaving factor defines the higher-order address "offset" by which addresses are incremented until the interleave "loop" is completed. Between loops, low-order addresses are incremented to the next physical word whereupon offset increments are used through the next loop (and so on). The value of n = 0 is reserved for null tuples and is niether allowed nor meaningful (n=0 => 20=1 => noninterleaved).*					

Assumptions:

The software interleave tuple must be associated with a Format Tuple (partition tuple set) that describes a homogeneous partition beginning and ending on a physical device boundary and hence referring to one and only one Device Info entry and associated Device Geometry Info entry within the Device Information Tuple and the Device Geometry Tuple respectively.

The DGTPL_HWIL field within the Device Geometry Info entry must equal 1, i.e. non-hardware interleaved.

The following provide the association of the Software Interleave Tuple and the implied "software interleave partition" with the Device Information Tuple, Device Geometry Tuple, and Format Tuple.

"software interleaved partition" base address = TPLFMT_OFFSET

"software interleaved partition" length = TPLFMT_NBYTES

Note - the size or length of the partition must be a multiple of the stride (see below).

*"software interleaved partition" interleave width = DGTPL_WBS * DGTPL_BUS*

*"software interleaved partition" stride = DGTPL_BUS * (size in bytes of a physical device)*

Note - the device size is determined from the Device Info entry of the Device Information Tuple. The Device Size Byte is found within table 5-15 of the PCMCIA 2.0 specification.

5.3.3 Standard Data Recording Formats

This standard allows great flexibility in adjusting the card format to meet specific requirements. For simplicity, this standard further specifies recommended formats for low-level data recording—formats which are expected to be commonly used. Higher level data formats will be specified in addition to this level.

- Generic — the bytes are recorded in 512-byte blocks with no error checking.The card's first data byte appears at byte address 512 (200H).
- Single-byte checksum format — the bytes are recorded in 512-byte blocks, with a separate region reserved for error-checking codes, and with a sector buffer.
- Two-byte Embedded CRC format — the bytes are recorded in 512-byte blocks, with each block followed by a 16-bit CRC.
- Raw- byte format — the bytes are recorded sequentially in an unblocked form.

In order to maintain a reasonable degree of interchangeability, this standard recommends that all Layer-2 conforming implementations be able to read and write generic-format cards.

When an implementation is presented with a card, whose format is not supported by that implementation, the implementation shall refuse to write on the card, except to reinitialize it. If the basic format is not supported by the implementation at all, the implementation shall return an error to applications whenever they attempt to access the card. The implementation may allow read-only access to a card whose basic format is supported, but whose error-detecting code is not.

Byte 2 (TPLORG_TYPE) specifies the type of data organization in use. The possible values of this byte are given in Table 5-57.

Table 5-57: Data Organization Codes

Code	Name	Description
0	TPLORGTYPE_FS	This partition contains a file system. The description field specifies the file system type and version.
1	TPLORGTYPE_APP	This partition contains application-specific information. The description field specifies the application name and version.
2	TPLORGTYPE_ROMCODE	This partition contains executable code images. The description field specifies the name and version of the organization scheme.
3-7FH		(Reserved for future standardization.)
80H-FFH	TPLORGTYPE_VS	This partition uses a vendor-specific organization. The contents of the description field are vendor-specific.

Bytes 3 through the end of the tuple (TPLORG_DESC) contain a NUL-terminated ASCII-text description of the organization. For file-system organizations, this field should specify the file-system type. This field shall contain only characters in the printing ASCII set, 020H through 07EH. (For international use, one or more CISTPL_ALTSTR tuples can follow this tuple.)

For DOS-file systems, TPLORG_TYPE shall contain TPLORGTYPE_FS, and TPLORG_DESC shall contain the string "DOS". For *version one* of the Flash file systems, TPLORG_TYPE shall contain TPLORGTYPE_FS, and TPLORG_DESC shall contain the string "Flash". *For version two of the Flash file system, TPLORG_TYPE shall contain TPLORGTYPE_FS, and TPLORG_DESC shall contain the string "FFS20". All TPLORG_DESC fields must be null terminated.*

The intent of this field is two-fold:

- For operating systems with sufficient flexibility, it allows the appropriate file-system driver to be selected based on the value of this field.
- If a card cannot be read due to software incompatibilities, a utility program can display to the user the contents of this field along with other card information. This would inform the user what kind of information is actually on the card.

5.5 System-Specific Standards (Layer 4)

Layer 4 of this standard specifies things that are only relevant in certain operating environments. At present, all layer-four standards are specific to the DOS environment. The following DOS-specific standards are defined:

- An interchange format for cards formatted with the DOS FAT-based file system (Section 5.5.1).
- A standard for directly-executable programs (see Section 6).
- A standard for interpreting older cards formatted without the Card Information Structure (Section 5.5.3).

5.5.1 Interchangeable Card Format

This standard would be of little use if it did not allow the free interchange of information between DOS systems. Rather than limiting all DOS implementations to a single format, this standard requires that all implementations support the following format, in addition to any other formats.

The Interchangeable card format has the following characteristics:

Layer 1 — the Card Information Structure shall contain at least a device-information tuple.

Layer 2 — the Card Information Structure shall contain the following tuples:

1. Level-2-information tuple, with the following fields set.
   - TPLLV2_COMPLY shall be 0.
   - TPLLV2_NHDR shall be 1, indicating that only one copy of the CIS is present.
   - TPLLV2_VSPEC8 and TPLLV2_VSPEC9 shall be zero.
2. A single format tuple. This tuple shall indicate that the partition uses a disk-like format with 512-byte blocks. It shall further indicate that the card uses no errror- detection code, that the EDC length is zero, and that the card's first data byte appears at byte 512, or higher. This tuple shall indicate that the partition covers all but the first 512 bytes of the card, and that there are (partition_size / 512) blocks in the partition.

In addition, the CIS may optionally contain the following tuples:

- A single geometry tuple. If present, this tuple shall contain information that matches the information presented in the boot block BPB.
- A single card-initialization date tuple.
- A single battery-replacement date tuple.

Layer 3 — the card shall contain a DOS-compatible file system. The boot sector shall be recorded in data-block zero (that is to say, starting at byte 512 of the card). The BPB in the boot sector shall describe the geometry of the file system in any convenient fashion. This standard recommends that geometry parameters be set to appropriate powers of two.

5.5.2 Execute In Place

The format for cards supporting direct execution of application programs from the card ("execute in place") is described in Section-6.

5.5.3 Interpreting Cards Without Card Information Structures

Some existing systems use RAM cards with a pseudo-floppy organization. Pseudo-floppy cards have the following format:

- A series of contiguous logical sectors, as viewed by MS-DOS.
- The BIOS sector addressing scheme (head, cylinder, sector) is mapped one-to-one to the logical sectors.
- The logical sectors are arranged exactly as in the case of a floppy disk, i.e. the first is the Boot Sector, then comes a variable number of File Allocation Table (FAT) sectors, then a number of Root Directory sectors, and finally the card is filled up with Data sectors.
- Sectors are a standard MS-DOS size: 128, 256 or 512 bytes with 512-byte sectors as the default, since this allows room for executable code in the first (Boot) sector.
- The Boot sector is defined exactly as for a floppy. It thus contains the BIOS Parameter Block (BPB), and, therefore, a definition of the number of bytes per sector, number of copies of the FAT, and so on.

Boot-Sector Structure

The boot sector would typically be 512 bytes if it were to contain bootstrap code as well as the BPB. Otherwise, it could be as small as 128 bytes. However, the first 30 to 50 bytes of the bootstrap sector are always recorded in a standard format.

The first three bytes of the boot sector are reserved for a short or a near jump:

- E9 XX XX

or

- EB XX 90

This gives us a simple way to detect a pseudo-floppy boot block. The BIOS Parameter Block would then follow.

The format of the boot-sector header is shown in Tables 5-35 and 5-36.

[Note:The information in these tables is controlled by DOS. The data formats are included here only for reference.]

Table 5-58: DOS Boot-Block Structure

Byte	7	6	5	4	3	2	1	0
0 .. 2	Short or near jump: 0E9H XX XX or 0EBH XX 090H.							
3 .. 0AH	System ID (OEM name and version) (8 bytes)							
0BH .. 0CH	Bytes per sector (2 bytes)							
0DH	Sectors per cluster							
0EH .. 0FH	# reserved sectors							
10H	Number of FATs							
11H .. 12H	# root directory entries							
13H .. 14H	# sectors in logical volume							
15H	Media descriptor byte							
16H .. 17H	# sectors per FAT							
18H .. 19H	# sectors per track							
1AH .. 1BH	number of heads							
1CH .. 1DH	# hidden sectors							

With DOS 4.0, and later versions, the following additional fields are defined:

Table 5-59: Extended BPB

Byte	7	6	5	4	3	2	1	0
1EH .. 1FH	# hidden sectors (most significant word)							
20H .. 23H	# sectors in logical volume (4 bytes)							
24H	Physical drive number							
25H	Reserved							
26H	Extended boot signature (29H)							
27H .. 2AH	Volume ID (binary) (4 bytes)							
2BH .. 35H	Volume label (11 bytes)							
36H .. 3DH	Reserved (8 bytes)							

The information in the BPB can be accessed by the device driver whether embedded in the ROM BIOS, external to the BIOS, or separately loaded as an MS-DOS Installable Device Driver. In this way, differing configurations of ROM BIOS-to-Logical Sector Mappings can be used. A card can have "multiple logical heads", for instance.

5.5.3.1 Handling Pseudo-Floppies in a Conforming System

Pseudo-floppies can easily be handled from within a conforming system, if the following procedure is followed during card-insertion processing:

1. Read the first byte of Attribute Memory. If it is 01H, process the CIS from Attribute Memory in the usual way. If all Metaformat information is present in Attribute Memory, or in Common Memory as specified by the AttributeMemory, then this is not a pseudo-floppy. If no CIS is present, or if a CIS is present in Attribute Memory, but no Layer-2 information is present, proceed to step 2.
2. Read the first few bytes from the card's Common Memory (starting at physical address 0) into a local buffer.
3. Compare the first five bytes of the buffer to the CIS link-target tuple signature (13H, xx, "CIS", where xx is a link value in the range [03H, FEH]). If they match, then this card has the CIS Metaformat structure in RAM (and probably has no Attribute Memory). Process by the usual rules. If they do not match, no CIS is present and proceed to step 4.
4. Compare the first three bytes of the buffer to the DOS boot block signature: 0E9H, XXh, XXH, or 0EBH, XXH, 90H. If it matches, assume that this is a DOS-format, pseudo-floppy. Extract the relevant geometry and block-size information from the BPB, assume that there is no error detection, and assume that the card consists of a single data partition encompassing the number of blocks indicated in the BPB.

5.6 Compatibility Issues

5.6.1 Buffer Pages

Some vendors use a buffer page to improve the reliability of memory cards in the face of power failures. This standard does not directly provide a means for specifying the location of the buffer page. Space can easily be reserved for a buffer page by proper adjustment of the values in the format tuple. If needed, a vendor-specific tuple can be added to specify the location of the buffer page within the partition.

5.6.2 Formatting Cards Under DOS

Use of a CIS does not require a special (non-DOS) format utility in the common case where the entire card is to be formatted as a DOS file system. For example, the memory card BIOS could determine all the relevant information (including the pseudo-disk geometry) using information that is passed to the BIOS format function. It could then transparently construct the CIS during the format operation. DOS is unaware of the existence of the CIS. When it reads block 0 of the disk, the BIOS returns the first user-accessible block.

For multi-format cards, a special format utility is required in order to get the proper partition information in the CIS.

SECTION - 6

EXECUTE IN PLACE (XIP)

EXECUTE IN PLACE (XIP)

6.1 Format, Size, Organization of Data in a PC Memory Card[1]

This specification describes the Metaformat tuples, data structures, driver architecture, and Application Programming Interface(API) for eXecute In Place (XIP).

6.1.1 LXIP and EXIP

Two types of XIP support are defined: LXIP and EXIP. LXIP refers to applications structured to operate in a 16KB paged-execution environment similar to that defined by the Lotus/Intel/Microsoft (LIM) 4.0 standard. EXIP refers to applications structured to operated in an Intel 80386 extended- addressing-mode-execution environment. These differences have no effect on the Metaformat, data structures, or driver architecture. They are only noticeable in the API described later in this document. There is significant difference in the hardware support required, and in the way applications are structured in the two environments. Discussion of how to construct XIP applications is outside the scope of this specification.

6.1.2 Card Partition Format and Size

The tuple data structures describe the format, size, and organization of various partitions within the memory space of a PC Memory Card. In the context of a PC Memory Card a partition is simply a region of memory within the card's address space.

6.1.3 File System Partitions

This specification assumes that an XIP partition is used only to store XIP applications and is not being used as part of a FAT or Flash file-system partition.

6.1.4 XIP Partition Identification

There are two tuples relevant to XIP. The "Format Tuple (CISTPL FORMAT)[2]" defines the card's data-recording format as well as the location and size of the card's associated memory region. The "Organization Tuple (CISTPL ORG)[3]" defines the organization of the data in a specific partition. This tuple must follow a format tuple to be associated with it.

The format tuple for an XIP partition should have a TPLFMT_TYPE field value of TPLFMTTYPE_MEM. An error-detection method may be specified. The TPLFMT_OFFSET and TPLFMT_NBYTES fields define the location and size of the XIP partition. The TPLFMT_FLAGS should be set to 0. The TPLFMT_ADDRESS field is not used, and should also be set to 0.

If the partition does not begin and end on a 16384 byte (16KB) boundary, portions of memory outside of the XIP application may be mapped into system address space during application execution. Also, if an XIP partition is not aligned on erase block or device boundaries, data outside of the XIP partition can be destroyed if the Erase XIP Partition function is invoked.

The "Organization Tuple (CISTPL ORG)" contains the field — TPLORG_TYPE — that identifies the type of organization associated with a specific partition. If the TPLORG_TYPE field is equal to the TPLORGTYPE_ROMCODE code, the partition contains XIP images. For a DOS-compatible XIP partition the TPLORG_DESC field should have a value of "DOS_XIP".

[1] The glossary in Appendix A defines many of the technical terms in this chapter.
[2] See Section 5.3.2.1.
[3] See Section 5.4.1.1.

[Note: Even though the TPLORGTYPE_ROMCODE suggests that the XIP partition is read-only, writing to areas of the partition is allowed when the card technology supports write access.]

6.1.5 XIP Partition Structure

Within the XIP partition, a format for the XIP images needs to be defined so that the XIP manager can locate the XIP images it is supposed to manage and map. Based on the size of the data structure used to describe each entry, and its offset within the first 16-Kbyte block of the partition, up to 511 XIP applications could reside within a single XIP partition. [Note: The XIP directory is not required to be 16Kbyte in size, but it must be located starting in the first 16-Kbyte block.]

The XIP partition is structured as follows:

An XIP header is located at the beginning of the XIP partition. The header consists of:

- 2-byte field defining the number of possible entries in the XIP directory,
- 4-byte partition serial number (similar to a DOS volume serial number),
- 2-byte field for data-structure-version number (0001H for this release),
- 24 bytes reserved (set to 0FFH)

The total length of the directory for an XIP partition can be determined by:

directory length = (number of directory entries) * 32 bytes.

The XIP partition is divided into an integer multiple of 16 Kbyte fixed-length "pages", where each 16-Kbyte page is aligned on 16-Kbyte boundaries. Each 16-Kbyte region is assigned a page number. The relationship between the page numbers and the byte offsets, relative to the beginning of an n-byte-long XIP partition, is partially listed in Table 6-1.

Table 6-1: Page Numbers & Relative Byte Offsets into an XIP Partition

16K Page Number	Byte Offset relative to the Beginning of an XIP Partition	Page Usage
0	00000H...03FFFH	XIP Header & Directory Structure
1	04000H...07FFFH	XIP Applications
2	08000H...0BFFFH	" "
3	0C000H...0FFFFH	" "
4	10000H...13FFFH	" "
5	14000H...17FFFH	" "
6	18000H...1BFFFH	" "
7	1C000H...1FFFFH	" "
.	.	" "
.	.	" "
.	.	" "
((n/4000H)-1)	(n-4000H)...(n-1)	" "

XIP directory entries are stored as an array of fixed-length structures. The layout of an XIP directory entry is similar to that of a DOS FAT-file-system-directory entry.

The XIP driver uses the XIP directory to determine the number of applications which are present, their names and extensions, time and date of creation, and their locations within the partition. The XIP directory starts immediately after the XIP header.

An XIP application occupies some number of contiguous 16-KB pages. All XIP application entries are allocated from the beginning of the XIP directory. That is, all directory entries are allocated contiguously, and all XIP pages are allocated from the beginning of the partition'sa available XIP memory pages.

The XIP driver can determine the next non-allocated memory page, and directory entry, by sequentially searching the XIP directory and finding the first entry that has never been allocated (i.e. XIP_STATUS = 0xxxxx111b). This directory entry is available for allocation, and the previous directory entry contains the information necessary to determine the next available XIP memory page.

The XIP driver may need to read the directory, at driver initialization time, and whenever a card with an XIP partition is installed in a slot. This is in order to determine any information the driver may need to operate properly. The XIP driver will also read the "directory" at driver run time in order to support XIP functions that access the XIP directory. If the type of memory making up the XIP partition permits it, the XIP driver may be able to add new XIP applications into the partition. ROM XIP partitions cannot support adding new XIP applications, but Flash or EPROM XIP partitions could support this functionality.

Each XIP directory entry has the following format:

Table 6-2: XIP Directory Entry Structure

Byte	Name
0...7	XIP_NAME
8...10	XIP_EXT
11	XIP_STATUS
12...13	XIP_APP_BEGIN
14...15	XIP_APP_OFFSET
16..21	XIP_RESERVED
22...23	XIP_CREATION_TIME
24...25	XIP_CREATION_DATE
26...27	XIP_FIRST_APP_PAGE
28...31	XIP_SIZE

Bytes 0-7 (XIP_NAME):

Specifies an ASCII string that is the primary name of this XIP application. The format of this name is the same as the DOS primary-file name, that is, a maximum of eight (8) characters. If the name is less than eight (8) characters it must be padded with blanks (20H).

Bytes 8-10 (XIP_EXT):

Specifies an ASCII string that is the name extension of this XIP application. The format of this name is the same as the DOS file-name extension, that is, a maximum of three (3) characters. If the extension is less than three (3) characters it must be padded with blanks (20H).

Byte 11 (XIP_STATUS):

Specifies the status of this XIP directory entry (see Table 6-3). This XIP status bears absolutely no relationship to the DOS file-attribute byte. Values for the status bits are chosen so that an XIP directory can be updated in media like Flash memory, without first erasing and then re-recording it. Flash memory has unique constraints placed on how bit values in it may change.

If XIP_STATUS has a value of FFh, the previous XIP directory entry was the last one in the XIP partition. New entries in the XIP directory can be made at this point in the directory. If the first byte in the first directory entry has a value of 0FFH, the XIP directory is completely empty. If the XIP directory is completely empty, the first page occupied by the directory may be partially available, the last page may be partially available, and all remaining pages of the partition are available for allocation. Whether there are any partially-available pages is determined by the total size and location of the partition, and the size of the XIP directory. The size and physical location of the XIP partition, within the card's physical-address space, is determined by the tuple data structures.

If XIP_STATUS indicates a deleted entry, the XIP_FIRST_APP_PAGE and XIP_SIZE are still necessary for managing which pages may be allocated to new XIP applications that are being copied into this partition. Also note that the XIP pages that have been deleted are not reusable because they have not been erased. When an entry is marked deleted, the name and extension fields of the directory entry must be ignored, since they are not required to be cleared.

Table 6-3: XIP Directory Entry Status Bit Flag Definitions

Bit	Definition
0..2	XIP allocation status 111 XIP directory entry is free to be used to store the next XIP directory entry. The previous XIP directory entry is the last valid entry in the XIP directory. 011 XIP directory entry and the contents of its associated XIP application are in the process of being created. Since this XIP directory entry has not yet been completely processed, an XIP loader must not load and execute this application because the XIP application code has not been completely copied! However, the XIP_FIRST_APP_PAGE and XIP_SIZE fields are valid for this directory entry. An XIP utility program must "close" the XIP directory entry before an XIP loader can load an XIP application. 001 XIP directory entry is allocated and contains an XIP application that may be executed.000 XIP directory entry was allocated but has been deleted. It no longer contains a valid XIP application that may be executed. An XIP loader should not load and execute it! The fields XIP_FIRST_APP_PAGE and XIP_SIZE contain valid data that must be used when searching for the next free page within the XIP partition.
3	LXIP/EXIP application. 0 - The application is structured for LXIP. 1 - The application is structured for EXIP.
4...7	Reserved for future use. Must be all ones.

Bytes (12-13) XIP_APP_BEGIN

Specifies the offset in the first page for the application's entry point. The offset should be located on a 16-byte boundary.

Bytes (14-15) XIP_APP_OFFSET

Specifies the offset in the first page of the application to the application's first byte. The offset should be located on a 16-byte boundary.

Bytes (16-21) XIP_RESERVED

These bytes are reserved for future use. All reserved bytes must be set to a value of FFH.

Bytes (22-23) XIP_CREATION_TIME

Specifies the time that this XIP directory entry was created, or added, to the XIP partition. For media such as Flash memory, it is not possible to "update" an XIP directory entry without first erasing the entire XIP partition. Therefore, the time only refers to the time that the XIP application was added to the XIP directory. The format of the create time is DOS compatible and is described in Table 6-4.

Table 6-4: Creation Time Bit Field Definitions

Bits	Definition
0...4	Binary number of 2 second increments (0-29, corresponding to 0-58 seconds)
5...10	Binary number of minutes (0-59)
11...15	Binary number of hours (0-23)

Bytes (24-25) XIP_CREATION_DATE

Specifies the date that this XIP directory entry was created, or added, to the XIP partition. For media such as Flash memory, it is not possible to "update" an XIP directory entry without first erasing the entire XIP partition. Therefore, the date only refers to the time that the XIP application was added to the XIP directory. The format of the create date is DOS compatible and is described in Table 6-5.

Table 6-5: Creation Date Bit Field Definitions

Bits	Definition
0...4	Day of month (1-31)
5...8	Month (1-12)
9...15	Year (relative to 1980)

Bytes (26-27) XIP_FIRST_APP_PAGE

Specifies the first 16-Kbyte page, within an XIP partition, that is allocated to this XIP application. Pages allocated to an XIP application should be allocated contiguously within the XIP partition. Based on the way pages are allocated to XIP applications, one must locate the last valid entry in the XIP directory structure to find the next free page. This may be done by searching through the XIP directory from the beginning, and inspecting the XIP_STATUS flags until the last entry is located. Note that in addition to entries that have not been closed, allocated and deleted entries are valid and possible. The next free page is determined by applying the following formula to the last valid entry in the XIP directory.

new_page_offset =(XIP_SIZE + XIP_OFFSET) mod 4000H

next_free_page = XIP_FIRST_APP_PAGE + INT ((XIP_SIZE + XIP_OFFSET)/4000H)

Pages from the next_free_page to the partition's last page number can be allocated to new XIP applications. If page-aligned allocations are desired, and if new page _offset is not zero, then the indicated page is a partially-allocated page, and the next page should be used.

[Note: It is possible to pack multiple (small) applications into a single page and have unique directory entries for each application.]

Bytes (28-31) XIP_SIZE

Specifies the size, in bytes, of this XIP application. This size must include any padding required to achieve any application-specific alignment. The size can be zero to allow an entry to simply be a "label".

6.2 XIP Device Driver Architecture

The XIP device-driver functionality is split between two device drivers: a high-level driver (XIP.SYS), and a low-level driver (PCMCIA.SYS). The high-level driver simply manages the data in the XIP partition and provides all the services required by an XIP application. Essentially, this driver provides the XIP API services to the XIP application. Another lower-level driver (PCMCIA.SYS) provides the hardware-related services to the high-level driver when it actually needs to manipulate the mapping hardware (i.e. mapping pages, saving or restoring mapping contexts, etc.)

By removing the hardware dependencies, the high-level driver becomes generic and can be used on any type of system with XIP capabilities. A system software company could then create this driver and license it to various system hardware OEMs. That would relieve them from having to create such a driver. The OEM can then concentrate on developing the relatively simple low-level driver specific to their hardware. Nothing prevents a systems OEM from supplying both levels of functionality within the high-level driver.

The overhead imposed by this dual-layer-driver approach is not expected to cause perceptible performance degradation in XIP applications.

6.2.1 Migration Path of Drivers

Implementations of this architecture are expected to migrate into BIOS functions over time. The two-level-driver architecture will initially be implemented as DOS-loadable drivers. Some manufacturers may initially provide both drivers as a single unit. This type of implementation will provide PCMCIA support on existing desktop systems using add-in, interface-cards with external card sockets.

Next, the PCMCIA.SYS driver is expected to be included in the BIOS for new systems. This could result in more optimized implementations with better memory utilization. It is even possible for some implementations to include the XIP driver as a built-in capability. There should be no impact to applications that run on these systems because they will not be able to distinguish between a loaded driver versus a BIOS driver.

6.2.2 High Level Device Driver Functions

Most of the functions in the XIP API perform operations that require hardware-specific knowledge. That is, they require information about a system's hardware configuration, and knowledge of how the system's XIP-mapping hardware operates. By defining a high- and low-level architecture, this hardware dependency can be removed from the high-level driver, and moved into the relatively simple low-level driver.

6.2.3 Low Level Device Driver Functions

The functions required by the low-level-device driver are defined in the Socket Services specification.

6.2.4 Sharing the Hardware Interface Between Device Drivers

It is possible that other device drivers ,unrelated to the XIP driver, will need to access the memory-mapping-interface hardware that maps memory on the card into the system's address space. It should be obvious that a memory-mapping interface is required for XIP. However, a memory-mapping interface would be perfectly acceptable for a disk device-driver as well. The same hardware interface could be used for XIP driver, a DOS FAT-file-system driver, and a DOS Flash-file-system driver.

Consequently, it is very important for system designers to provide the ability to read, as well as write, the state of their XIP-mapping hardware. Write-only, memory-mapping-hardware registers are not acceptable. This is because a disk-device driver would have to save and restore the state of the XIP-mapping hardware, before and after a disk access, as the mapping hardware may be in use by an XIP driver and XIP application. [Note: An XIP driver should not do saves/restores of its mapping hardware between XIP function calls!]

6.3 LIM 4.0 Compatibility

A LIM 4.0 driver and an XIP driver perform similar operations on memory. They both map regions into the system's memory space of any private memory space that they manage. Since XIP drivers and applications must be compatible with LIM drivers and LIM applications, an XIP driver must determine two things:

1. whether a LIM driver is installed in the system, and,
2. what regions of memory the LIM driver maps.

The description of an integrated LIM and XIP driver is beyond the scope of this specification.The following are the steps for working out LIM and XIP compatibility.

1. The LIM driver must be loaded first.
2. The XIP driver must check for the presence of a LIM driver. The XIP driver must not use the areas of memory mapped by the LIM driver. The XIP driver can determine what these areas are by querying the LIM driver using a LIM function that returns this information.
3. The XIP driver should not use system-memory areas that are occupied by RAM, ROM, or BIOS shadow areas. The XIP driver should use typical industry practices, such as checking for BIOS extension signatures, in detecting the presence of ROMs and RAMs. (A complete discussion of these practices is beyond the scope of this specification.)
4. XIP driver should use command-line parameters to exclude or include regions of system memory that are appropriate for the XIP driver to use for XIP-application mapping. The command- line parameters should override any algorithms that the XIP driver uses to "automatically" determine the presence of RAM or ROM.

6.3.1 Device Driver Load Order

The XIP.SYS and PCMCIA.SYS device drivers would be loaded like any other device driver during CONFIG.SYS processing.

For the purposes of LIM compatibility, the XIP.SYS and PCMCIA.SYS drivers must be loaded after a LIM driver that is being loaded during CONFIG.SYS processing.

For the XIP drivers, the PCMCIA.SYS driver must be loaded before the XIP.SYS driver. During XIP.SYS initialization, information about the system's mapping capabilities, which can only be provided by the PCMCIA.SYS driver, is required by the XIP.SYS driver.

6.4 XIP Loader

The XIP loader invokes the functions supplied by the XIP driver. The function of the XIP loader is to look up, map, and start the XIP application execution.

There are several possibilities for implementing a loader. The XIP loader could be supplied by the application developer or the system manufacturer. The goal is for users to invoke XIP applications on DOS PC platforms in the same way that they invoke non-XIP applications. The following outlines a possible loader implementation which achieves this goal. [Note: Other types of systems could still make use of the XIP definition with different loader implementations.]

First assume the following:

1. A file-system partition has been created on the same card that contains the XIP partition.
2. This file system contains a loader stub for each XIP application. If applications are added to the XIP partition, corresponding loader-stub entries are made to the file-system partition.
3. There are two environment variables — SLOTPATH and CURSLOT — that define the order in which to search slots for XIP partitions (similar to drive order for file searches), and define the current "logged" slot (similar to current directory).

In this environment, the same loader stub can be used for all applications and behaves as follows:

1. When the loader is executed, it determines the name with which it was invoked and searches the XIP partition for an entry of the same name. The search starts with an XIP partition (if one exists) in the current slot (defined by CURSLOT) and then searches for other XIP partitions in the order specified by SLOTPATH.

2. If no matching application name is found, an error is returned. Otherwise, the first page of the application is mapped and execution is transferred to the entry-point offset, as specified in the directory entry. If the mapping request fails, an appropriate error message can be displayed.

[Note: if an extended XIP loader is used instead, it could map in the whole application using the information included in the directory entry and then transfer control.]

6.5 XIP Applications Programming Interface

6.5.1 XIP API Callback Interface

The XIP device driver uses a procedure-call interface rather than a software-interrupt interface. The callback interface, being private, means that other software, not directly related to the operation of the XIP device, will not be chained into the execution path. The benefit of this interface is that there are no software interrupt-chaining-compatibility problems.

6.5.2 Initializing the XIP Interface

In order for an XIP application to use the XIP device driver, the XIP application needs to know the entry point of the XIP services within the device driver. The process of determining whether the XIP driver is installed, and getting its entry point, is outlined in the following steps.

1. Issue a DOS "open read-only mode." This function requires a far pointer to the ASCIIZ string containing the device name to open. In this case, the device name is actually the internal name found in the XIP device driver's header. The pointed-to ASCIIZ string should have the following format:

   XIP_device_name DB "XIP$$$$$", 0

   If DOS does not return an error status, one can assume that either a device with the name "XIP$$$$$" is installed, or a file with this name is on the current disk drive. Proceed to step 4, otherwise, proceed to the next step.

2. If DOS returned a "too many open files" status, one can modify one's application so that it opens the XIP device before it opens any other files. The XIP handle is not used after the entry point is obtained. If this was not the error one's application received, proceed to the next step.

3. If DOS returned a "file/path not found", the XIP-device driver is not installed. If one's application requires the XIP-device driver, there is only one way to correct the problem. The user must install the XIP-device driver, modify the CONFIG.SYS file to reflect the installation, and reboot the system before proceeding. One's application cannot proceed further.

4. Issue a DOS IOCTL "get device data" using the handle obtained in step 1. This function returns device data that allows one to determine whether XIP is a device or a file.

   If DOS returns any error status, one may assume that the XIP device driver is not installed. The user will have to follow the procedure outlined in step 3 to correct the problem.

5. Check that "XIP$$$$$" is a device and not a file with the same name. The device data returned by the previous DOS function contains the ISDEV bit (DX bit 7). If the ISDEV bit is a 1 then "XIP$$$$$" is a character device and not a file. If ISDEV is bit is a 0 then "XIP$$$$$" is a file and there is no XIP-device driver installed. The user will have to follow the procedure outlined in step 3 to correct the problem. Also, the file named XIP should be renamed so that the user may access it after the XIP driver is installed. This should be an extremely rare situation.

6. Issue a DOS "IOCTL read" using the handle obtained in step 1 for a maximum of 4 bytes.

   If DOS returns any error status, or the driver does not transfer the specified number of bytes, one may assume that the XIP-device driver is not a compliant driver. The user will have to follow the procedure outlined in step 3 to correct this problem.

7. Save the device driver entry-point address returned by the "read".

8. Issue a DOS "close" command using the handle obtained in step 1. Doing so frees up the handle allocated by the original "open". The handle is not used again.

The following procedure is an example of the technique outlined in this section.

```
;-----------------------------------------------------------------------------------------;
;   open_XIP_driver;                                                                      ;
;   The procedure verifies that the XIP driver is installed in the system and ;
;   returns a handle so that driver IOCTLs may be done if it is present.                 ;
;   If XIP driver is installed                                                            ;
;       CARRY CLEAR                                                                       ;
;       (bx) = handle for XIP device driver get/set calls                                 ;
;   else                                                                                  ;
;    CARRY SET                                                                            ;
;-----------------------------------------------------------------------------------------;

open_XIP_driver                            proc
;-----------------------------------------------------------------------------------------;
;   Open the XIP device.;                  ;
;-----------------------------------------------------------------------------------------;
    mov         dx, offset XIP_device_name   ; (ds:dx) = far ptr to device name string
    mov         ax, 3D00h                    ; (ax)    = open read-only function
    int         21h                          ; issue device read-only open
    jc          oXd_02                       ; error during device open

;-----------------------------------------------------------------------------------------;
;   Get the info flags for the XIP handle. ;
;-----------------------------------------------------------------------------------------;
    mov         bx, ax                       ; (bx) = handle returned by open
    mov         ax, 4400h                    ; (ax) = IOCTL get device data function
    int         21h                          ; issue get device data IOCTL
    jc          oXd_01                       ; error during IOCTL get device info

;-----------------------------------------------------------------------------------------;
;   Test the ISDEV bit in the device info flags.;
;-----------------------------------------------------------------------------------------;
    test        dx, 0080h                    ; (dx) = file or device data
    jz          oXd_01                       ; XIP is a file, NOT a device

;-----------------------------------------------------------------------------------------;
;   XIP driver is installed in this system.                                               ;
;   Return:                                                                               ;
;   (bx) = XIP driver handle.                                                             ;
;   (CARRY CLEAR) to indicate that the XIP device is installed.                           ;
;-----------------------------------------------------------------------------------------;

    clc
    ret
;-----------------------------------------------------------------------------------------;
;   XIP driver is not installed in this system.                                           ;
;   Close the file named XIP$$$$$.                                                        ;
;-----------------------------------------------------------------------------------------;
```

```
oxd_01:                                          ; (bx) = handle returned by open
    mov ah, 3Eh                                  ; (ah) = close function
    int 21h                                      ; close XIP$$$$$ file/driver
;---------------------------------------------------------------------------------------;
;   XIP driver is not installed in this system.                                        ;
;   Return:                                                                             ;
;   (CARRY SET) to indicate that the XIP device is not installed.                      ;
;---------------------------------------------------------------------------------------;

oXd_02:
    stc
    ret
open_XIP_driver                                  endp

;---------------------------------------------------------------------------------------;
;   XIP driver name.                                                                    ;
;---------------------------------------------------------------------------------------;
    XIP_device_namedb                            ”XIP$$$$$”, 0
```

6.5.3 XIP IOCTL References

The XIP device driver requires that applications use IOCTLs to get, and/or set, the entry point for the XIP device driver. This allows an arbitrary chain of applications to monitor or patch the device driver. The entry point of the XIP device driver is actually the procedure that an application calls whenever it needs to call the XIP API (Application Programming Interface).

6.5.4 IOCTL Read (Get Current XIP API Entry Point)

The DOS "IOCTL read" function (INT 21H, function 4402H) is used to obtain the XIP API entry point. This function will read, into a buffer supplied by the application, a dword pointer supplied by the XIP driver. The dword pointer in the buffer is a far pointer to a far pointer to the XIP API. All applications needing to use the XIP API must obtain this entry point before they can make an XIP API call.

The following example builds on the previous example and demonstrates how an application obtains the XIP API entry point.

```
;  ;
;  Get the current XIP API entry point.                                        ;
;  If XIP API services are available                                           ;
;      CARRY CLEAR;
;      (bx)           = handle for future XIP device driver get/set calls;
;      (XIP_callback) = far pointer to far pointer to the XIP API              ;
;  else;
;      CARRY SET;
;  ;

get_XIP_callbackproc
   call       open_XIP_driver            ; check for the XIP driver & open it
   jc         gXc_02                     ; XIP driver not installed

;  ;
;  Get the XIP API entry point.;
;  ;
                                         ; (bx) = XIP driver handle returned by open
   mov        dx, offset XIP_callback    ; (ds:dx) = far ptr to XIP callback buffer
   mov        cx, 4                      ; (cx) = # bytes to transfer (dword size)
   mov        ax, 4402h                  ; (ax) = IOCTL read device data
   int        21h                        ; issue IOCTL read device data
   jc         gXc_01                     ; error during IOCTL read device data

;  ;
;  Verify that only the XIP API entry point was transferred.;
;  ;
   cmp        ax, 4                      ; (ax) = # bytes actually transferred
   jne        gXc_01                     ; driver did not transfer the specified # of bytes
;  ;
;  XIP API services are available.;
;  Return:;
;     (XIP_callback) = far pointer to far pointer to the XIP API.;
;     (CARRY CLEAR) to indicate that the XIP API services are available.;
;  ;
   clc
   ret


;  ;
;  Close the XIP device.;
;  ;
gXc_01:; (bx) = handle returned by open_XIP_driver call
   mov        ah, 3Eh                    ; (ah) = close function
   int        21h                        ; close XIP device

;  ;
;  XIP API services are not available.;
;  Return:;
;      (CARRY SET) to indicate that the XIP API services are not available.;
;   ;
gXc_02:
    stc
    ret
get_XIP_callbackendp


;   ;
;   XIP callback storage.;
;   ;
    XIP_callbackdd                           ?
```

6.5.5 IOCTL Write (Set New XIP API Entry Point)

The DOS "IOCTL write" function (INT 21H, function 4403H) is used to set a new XIP API entry point. This function will write a dword pointer, supplied by the application, to the XIP driver. This dword pointer is a far pointer to the new XIP entry procedure. The function provides an XIP utility, or another device driver that needs to trap XIP API accesses, with the ability to chain into the XIP API's path of execution.

If one is creating an XIP utility that absolutely must chain into the XIP API, remember to restore the original XIP entry point before one's utility exits back to DOS. If one does not, and one's code exits, the next application that attempts to use the XIP API will probably hang the users system.

The following example builds on the previous examples and demonstrates how an application sets a new XIP API entry point.

```
;   ;
;   Get the current XIP API entry point and set a new XIP API entry point.;
;   If XIP API services are available;
;      CARRY CLEAR;
;      (bx)              = handle for future XIP device driver get/set calls;
;      (XIP_callback)    = far pointer to far pointer to the XIP API;
;      (old_XIP_ent_pt) = address of the current XIP API entry point.;
;      (new_XIP_ent_pt) = address of the new XIP API entry point.;
;   else;
;      CARRY SET;
;   ;
set_XIP_callbackproc
;   ;
;   Open the XIP driver and get the XIP callback.;
;   ;
    callget_XIP_callback; get XIP callback
    jc  sXc_01  ; could not get the XIP callback

;   ;
;   Save the address of the current XIP API entry point so that it can be restored;
;   later. The example assumes that the old XIP entry point is accessible via the;
;   example code segment.;
;   ;
    les         di, XIP_callback             ; (es:di) = far ptr to far ptr to XIP API entry point
    les         di, dword ptr es:[di]        ; (es:di) = far ptr XIP entry point address
    mov         word ptr cs:old_XIP_ent_pt[0], di
    mov         word ptr cs:old_XIP_ent_pt[2], es   ; (old_XIP_ent_pt) = current XIP entry point address

;   ;
;   Initialize a far pointer in a buffer so that it points to the new;
;   XIP API entry point.;
;   ;
    mov         word ptr new_XIP_ent_pt[0], offset XIP_trap
    mov         word ptr new_XIP_ent_pt[2], cs      ; (new_XIP_ent_pt) = new XIP entry point address
;   ;
;   Send the new XIP API entry point to the XIP driver.;
;   ;
                                             ; (bx) = handle returned by get_XIP_callback
    mov         dx, offset new_XIP_ent_pt    ; (ds:dx) = far ptr to new XIP entry point buffer
    mov         cx, 4                        ; (cx) = # bytes to transfer (dword size)
    mov         ax, 4403h                    ; (ax) = IOCTL write device data
    int         21h                          ; issue IOCTL write device data
    jc          gXc_01                       ; error during IOCTL read device data
```

```
;   ;
;   New XIP entry point has been set.;
;   Return:;
;      (bx) = handle for future XIP device driver get/set calls;
;      (CARRY CLEAR) to indicate that the XIP API services are available.;
;   ;
    clc
    ret

;   ;
;   XIP API services are not available.;
;   Return:;
;      (CARRY SET) to indicate that the XIP API services are not available.;
;   ;
sXc_01:
    stc
    ret
set_XIP_callbackendp


;   ;
;   New XIP entry point storage.;
;   ;
    new_XIP_ent_ptdd?

;   ;
;   Old XIP entry point storage.  This example assumes that this pointer resides in the  ;
;   same CODE segment as do the set_XIP_callback and XIP_trap procedures.;
;   ;
    old_XIP_ent_ptdd?
```

6.5.6 Chaining into the XIP API

The following example builds on the previous examples and demonstrates how an XIP utility would properly chain into the XIP API. The hypothetical example provided assumes that the original XIP driver cannot either write or erase the special devices the XIP_trap code supports. However, the original driver is capable of doing all other functions. Therefore, the example inspects the function codes passed to the XIP API and traps only erase and write functions rather than permitting the original XIP driver to do them. All other function will be passed on through.

The example also assumes that set_XIP_callback has been called and has completed successfully.

```
;;
;   This example code simply checks to see if the XIP API code passed to the XIP driver performs   ;
;   writes or erases. If it does, it services the erase or write. If it doesn't it chains through;
;   to the original XIP API entry point.;
;   ;

XIP_trapprocfar
;   ;
;   Trap the three XIP functions that perform writes or erases.;
;   ;
    cmp        ah, XIP_Add                  ; (ah) = XIP function code
    je         Xt_01                        ; trap Add XIP Directory Entry

    cmp        ah, XIP_Copy
    je         Xt_02                        ; trap Copy XIP Directory Entry

    cmp        ah, XIP_Erase
    je         Xt_03                        ; trap Erase XIP Partition

;   ;
;   Chain into the original XIP API entry point.;
;   ;
    jmp        dword ptr cs:old_XIP_ent_pt

;   ;
;   Your special trap code continues here.;
;   ;
Xt_01:.
    .
Xt_02:.
    .
Xt_03:.
    .
;   ;
;   Your trap code has finished its work.;
;   ;
XIP_trap_OK:
    clc
    ret                                     ; CARRY CLEAR indicates operation passed

XIP_trap_err:
    stc                                     ; CARRY SET indicates operation failed
    ret                                     ; (ah) = error status of operation
XIP_trap       endp
```

6.5.7 Calling the XIP API

As the previous sections discussed, the XIP_callback is really a far pointer to a far pointer. Calling the XIP API from one's application involves a simple two-step process. First, use an LDS or LES instruction to obtain the contents of the XIP_callback returned from the example get_XIP_callback procedure. Second, call the address pointed to by the address pointed to by the LDS or LES. This address is the XIP API entry point.

Again, note that the XIP_callback is a pointer to a pointer. It does not contain the address of the XIP API entry point, but rather a pointer to a location that does contain this entry point. This extra level of indirection adds flexibility in being able to both chain and unchain into an XIP driver.

As an example, suppose the application wanted to get the XIP API version.

```
;   ;
;   Assume that the XIP_callback was correctly initialized by the;
;   get_XIP_callback that you would have done only once at the beginning of;
;   your programs start-up code.;
;   ;
    mov         ah, 80h                         ; (ah) = get XIP version function
    lds         si, XIP_callback                ; (ds:si) = far ptr to far ptr to XIP API entry point
    call        dword ptr [si]                  ; call the XIP API. Note the indirect far call!
    jc          process_XIP_API_err             ; an XIP error status was returned-don't ignore it!


;   ;
;   The XIP API call completed. Your code continues here.;
;   ;
    .                                           ; (al) = XIP version
    .
    .

process_XIP_API_err:
;   ;
;   The XIP API call failed. Your error processing code continues here.;
;   ;
    .                                           ; (ah) = XIP error status
    .
    .
```

Note that this is exactly the same operation that one's application would do to call the XIP API, even if it, or another driver, had chained itself into the XIP API's execution path. Nothing special need be done by one's application to account for other drivers being chained into the XIP API.

6.6 XIP API Functions

The following functions comprise the entire set of XIP functions that are required to be a fully compliant XIP version 1.0 driver.

6.6.1 Get XIP Version (Both)

Purpose:

This function returns the version of the XIP driver installed in the system. An application uses this function to determine if the set of XIP functions it requires are supported by this XIP driver.

Calling Parameters:

AH = 80H   Get XIP Version function.

Results:

These results are valid only if the status returned is PASSED.

AL = version number

Contains the XIP driver's version number in binary coded decimal (BCD) format. The upper four bits contain the integer digit of the version number. The lower four bits contain the fractional digit of version number. For example, version 1.0 is represented like this:

```
0001  0000
  \    /
   1. 0
```

When checking for a version number, an application should check for a version number or greater. Vendors may use the fractional digit to indicate enhancements or corrections to their XIP drivers. Therefore, to allow for future versions of XIP drivers, an application should not depend on an exact version number.

AH = type(s) of XIP functions supported by this driver.

The rest of the function descriptions in the API are labeled with Write, LXIP, EXIP, or "Both" to indicate when they are supported by this driver. AH returns with bits set to indicate which functions this driver supports.

Reserved(0) bits 7..3	Write Functions	EXIP Functions	LXIP Functions

LXIP Functions = 1: Functions labeled with LXIP or Both are supported by this driver,

EXIP Functions = 1: Functions labeled with EXIP or Both are supported by this driver,

Write Functions = 1: Functions 8BH to 8FH are supported by this driver.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FFH	The function failed. Cause unknown.

6.6.2 Get XIP Mappable Segments (LXIP)

Purpose:

This function returns an array containing the segment address for each mappable XIP region in a system. The array is sorted in ascending segment order.

Calling Parameters:

AH = 81H	Get XIP Mappable Segments function.

ES:DI = mappable_XIP_segment_array_ptr

Contains a far pointer to an application-supplied memory area where the XIP driver will copy the array.

Results:

These results are valid only if the status returned is PASSED.

ES:DI = mappable_XIP_segment_array_ptr

Contains a far pointer to an application-supplied memory area where the XIP driver will copy the array. The first word of the memory area is the maximum number of entries allowed in the array. The remainder of the memory area is the array. The driver fills in the array. Each entry in the array is a single word. [Note: An 8-entry array will probably provide sufficient space in most cases.]

```
segment_struct                      STRUC
   length                   DW   ?
   mappable_XIP_segment     DW   ?
segment_struct                      ENDS
```

Each member is a word which contains the segment address of a mappable XIP segment in the system. Each mappable segment is 16-Kbytes long. The array entries are sorted in ascending segment address order.

CX = desired number of entries in the mappable_XIP_segment_array

If CX is greater than the maximum array size, a larger array is required. In this case, the provided array only has the first part of the information.

Registers Modified:

AX, CX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FFH	The function failed. Cause unknown.

6.6.3 Get XIP Partition IDs (Both)

Purpose:

This function returns an array of XIP partition IDs currently accessible in the system. XIP partition IDs are byte tokens used to uniquely identify each XIP partition. Token values are implementation specific and are only valid after this function is called.

Partition IDs are assigned at initialization time by the XIP driver. These IDs are only valid during a given system boot. That is, IDs can change from boot to boot. Normally IDs will not be reassigned to a different partition once they have been assigned to a specific partition. However, the Disable Partition ID function can be used to allow a partition ID to be reassigned.

This function will not return a partition ID that has been disabled. However, it can return new partition IDs that correspond to XIP partitions on a newly inserted card.

Calling Parameters:

AH = 82H	Get XIP Partition ID List function.

ES : DI = partition_IDs_ptr

Contains far pointer to an application supplied memory area where the XIP driver will copy the Partition IDs. The first byte contains the maximum length of the provided array.

```
IDs_struct                        STRUC
    length                  DB  ?
    IDs                     DB  ?
IDs_struct                        ENDS
```

Results:

These results are valid only if the status returned is PASSED.

ES : DI = partition_IDs_ptr

CX = desired length of list

If CX is greater than the maximum array size, a larger array is required. In this case, the provided array only has the first part of the information.

Registers Modified:

AX, CX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FFH	The function failed. Cause unknown.

6.6.4 Get XIP Handle Range (Both)

Purpose:

This function returns the range of handle values that the XIP driver manages. Handle values 8000h through, and including, 8FFFH are reserved for use by the XIP driver. The handle value returned by an XIP driver will have a value in this range.

The XIP driver assigns a unique handle value, in the specified range, to every XIP application present in the XIP directory. An XIP application uses a handle as a parameter whenever it calls the XIP driver to perform a map, unmap, or copy function.

Calling Parameters:

AH = 83H	Get XIP Handle Range function.

Results:

These results are valid only if the status returned is PASSED.

BX = First inclusive XIP handle

This is the value of the first handle managed by the XIP driver. Should be in the range 8000H through and including 8FFFH (unsigned word).

DX = Last inclusive XIP handle

This is the value of the last handle managed by the XIP driver. This value should be greater that the value returned in BX. Should be in the range 8000H through and including 8FFFH.

CX = Total handles

The maximum number of handles that this XIP driver is capable of supporting.

Registers Modified:

AX, BX, CX, DX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FFH	The function failed. Cause unknown.

6.6.5 Map/Unmap an XIP Handle's Pages (LXIP)

Purpose:

This function maps or unmaps one, several, all, or none of the logical pages associated with a handle into as many mappable segments as the system supports. Both mapping and unmapping pages can be done in the same invocation. Mapping or unmapping no pages is not considered an error. If a request to map or unmap zero pages is made, nothing is done and no error is returned. The segment map array passed to this function does not have to have any special order with respect to mappable_segment elements.

The function should allow mapping a specific logical page at more than one mappable segment address. This same "feature" is also a part of the LIM specification.

The handle needed for this function is obtained through the "Search for XIP Directory Entry". The total number of logical pages associated with this handle are also returned by these functions. The logical pages associated with a handle are all 16 Kbytes in length and are numbered zero relative to the beginning of the XIP application represented by the handle.

Calling Parameters:

AH = 84H Map/Unmap XIP Pages function.

AL = XIP partition ID

Contains the ID of the partition containing the XIP application pages to be mapped.

DX = handle

Contains the XIP handle associated with these pages.

CX = map_count

Contains the number of entries in the array. For example, if the array contained four pages to map or unmap, then CX would contain four.

DS:SI = far pointer to seg_map array

Contains a far pointer to an array of structures that contains the information necessary to map the desired pages. The array is made up of the following structure:

```
seg_map_struct                      STRUC
  log_page_number          DW   ?
  mappable_segment         DW   ?
seg_map_struct                      ENDS
```

.log_page_number

The first member is a word which contains the number of the logical page to be mapped. Logical pages are numbered zero relative with respect to the beginning of the XIP application with which they are associated. Therefore, the number for a logical page can range from zero to (number of logical pages allocated to the handle - 1).

If the logical page number is set to FFFFH, the mappable segment associated with it is unmapped rather than mapped. Unmapping a mappable segment makes it inaccessible.

.mappable_segment

The second member is a word which contains the segment address within the system-memory address space at which the logical page is to be mapped. This segment address must correspond exactly to a mappable segment address. The mappable segment addresses are available through the Get XIP Mappable Segments function.

Results:

CX = mapped count

Contains the count of the numbers of segments actually mapped whether the function succeeds or fails. If the function fails, the first CX segments requested are mapped and will eventually need to be unmapped by the application.

Registers Modified:

AX,CX

Status:

CARRY CLEAR PASSED

CARRY SET FAILED

AH = 83H The XIP driver could not find the specified XIP handle. The XIP driver does not have any information pertaining to the specified XIP handle. The program has probably corrupted its XIP handle.

AH = 8AH One or more of the logical pages to be mapped is out of the range of logical pages allocated to the XIP handle. The program can recover from this condition by mapping a logical page which is within the bounds for the specified XIP handle. When this error occurs, the only pages mapped or unmapped were the ones valid up to the point that the error occurred.

AH = 8BH One or more of the mappable segment addresses specified is not mappable, the segment address does not fall exactly on a mappable address boundary, or the map_count exceeds the number of mappable segments in the system. The program can recover from this condition by mapping into memory on an exact mappable segment address. When this error occurs, the only pages mapped were the ones valid up to the point that the error occurred.

AH = F8H The specified partition does not exist.

AH = FBH One or more of the logical pages specified is already mapped at a mappable segment. The previous logical page mapped at the specified mappable segment is still mapped. The new logical page specified is not mapped.

AH = FCH The card containing the XIP partition has changed since the last XIP API call.

AH = FFH The function failed. Cause unknown.

6.6.6 Get XIP Mapping Context Size (Both)

Purpose:

The Get XIP Mapping Context Size function returns the storage requirements for the array passed to the "Get XIP Mapping Context" and "Set XIP Mapping Context".

Calling Parameters:

AH = 85H	Get XIP Mapping Context Size function.

Results

These results are valid only if the status returned is PASSED.

AL = size_of_mapping_context

Contains the number of bytes that will be transferred to/from the memory area an application supplies whenever a program requests the Get or Set XIP Mapping Context functions.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FFH	The function failed. Cause unknown.

6.6.7 Get XIP Mapping Context (Both)

Purpose:

This function saves the mapping context for all XIP mappable memory regions by saving the state of the systems XIP mapping hardware in a destination array. The XIP application must pass a pointer to the destination array.

Calling Parameters:

AH = 86H	Get XIP Mapping Context function.

ES:DI = dest_page_map

Contains a far pointer to the destination array address in segment:offset format. Use the Get XIP Mapping Context Size function to determine the maximum size of this array.

Results:

These results are valid only if the status returned is PASSED.

ES:DI = dest_page_map

The array contains the state of the XIP mapping hardware. It also contains any additional information necessary to restore the XIP mapping hardware to its original state when the XIP application program invokes a Set XIP Mapping Context function. The content and size of this information is vendor specific.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.8 Set XIP Mapping Context (Both)

Purpose:

This function restores the mapping context for all XIP mappable memory regions by copying the contents of a source array into the systems XIP mapping hardware. The application must pass a pointer to the source array.

Calling Parameters:

AH = 87H Set XIP Mapping Context function.

DS:SI = source_page_map

Contains a far pointer to the source array address in segment:offset format. The XIP application must point to an array which contains the XIP mapping hardware state as returned by Get XIP Mapping Context. Use the Get XIP Mapping Context Size function to determine the maximum size of the array. The size and content of this information is vendor specific.

Results:

None.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = A3H	The contents of the source array have been corrupted, or the pointer passed to the function is invalid.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.9 Search for XIP Directory Entry (Both)

Purpose:

This function searches the XIP directory for an XIP application with a specific name. If the name is found, this function returns the handle associated with the name as well as the number of logical pages allocated to it.

The XIP handle returned by the XIP driver always specifies the same XIP application within the XIP partition. For example, if you searched the XIP directory six times for an existing XIP application named "WORDPROC.XIP", the XIP driver would return the same handle value each time because the position of the XIP application within the XIP directory structure had not changed.

An XIP handle is somewhat analogous to an index into an array of fixed length structures. The handle returned can be used by as many processes as need access to the XIP application.

Calling Parameters:

AH = 88H Search for XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory to be searched.

DS:SI = XIP_app_name

Contains a far pointer to an 8.3 string that contains the name of the entry to be searched for. Name and extension must both be padded with blanks (20H).

Results:

These results are valid only if the status returned is PASSED.

DX = XIP_app_handle

The value of the handle assigned to the XIP application specified.

CX = logical_page_count

The number of logical pages allocated to the XIP application. This value can be zero.

Registers Modified:

AX, CX, DX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = A0H	No corresponding handle could be found for the XIP application specified, i.e. the entry name was not found.
AH = A1H	The XIP application name is invalid.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.10 Get First XIP Directory Entry (Both)

Purpose:

This function returns the name, total allocated pages, and handle corresponding to the first active entry in the XIP directory in the specified partition.

Calling Parameters

AH = 89H   Get First XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory.

ES:DI = pointer to XIP_app_name

Contains a far pointer to a 32 byte buffer in RAM memory into which the XIP driver will copy the name of the XIP application if it can find one in the XIP directory.

Results:

These results are valid only if the status returned is PASSED.

DX = XIP_app_handle

The value of the handle assigned to the first XIP application in the XIP directory.

CX = logical_page_count

The number of logical pages allocated to the first XIP application in the XIP directory. This value can be zero.

ES:DI = pointer to XIP_app_name

Contains a far pointer to an area of RAM memory into which the XIP driver has copied the first XIP application's name if the XIP directory is not empty. This name buffer is formatted exactly the same as the XIP Directory Entry Sturcture (Table 6-2).

Registers Modified:

AX, CX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = A0H	No XIP application was found in the XIP directory. The XIP directory is empty.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.11 Get Next XIP Directory Entry (Both)

Purpose:

This function returns the handle, total allocated pages, and name corresponding to the next active entry in the XIP directory.

Calling Parameters:

AH = 8AH Get Next XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory.

DX = XIP_app_handle

The value of the handle returned from a "Get First XIP Directory Entry" call or from the previous "Get Next XIP Directory Entry" call. The XIP driver uses this handle value to begin the search for the next active entry in the XIP directory.

ES:DI = pointer to XIP_app_name

Contains a far pointer to a 32 byte area of RAM memory into which the XIP driver will copy the name and other information of the XIP application if it can find another active entry in the XIP directory.

Results:

These results are valid only if the status returned is PASSED.

DX = XIP_app_handle

The value of the handle assigned to this XIP application in the XIP directory.

CX = logical_page_count

The number of logical pages allocated to this XIP application in the XIP directory. This value can be zero.

ES:DI = pointer to XIP_app_name

Contains a far pointer to an area of RAM memory into which the XIP driver has copied this XIP application's information. The structure is defined in Table 6-2 of this section.

Registers Modified:

AX, CX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = 83H	The handle passed to the XIP driver is invalid. The XIP driver doesn't have any information pertaining to the specified XIP handle. The program has

probably corrupted the XIP handle it obtained from the previous "Get First XIP Directory Entry" or "Get Next XIP Directory Entry" calls.

AH = A0H	No more entries were found in the XIP directory.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.12 Add XIP Directory Entry (Both) (Write)

Purpose:

This function allows an XIP utility to insert a new XIP application into the XIP directory. The utility would use this function to name the XIP application and allocate space for it within the XIP partition. The utility would subsequently map the pages which had been allocated to the XIP application into memory and then copy the new XIP application into them. Not all media will support this operation. Specifically, only programmable devices allow the creation, copying, and deletion of XIP applications. This function is only supported by a driver that returns from Get XIP Version with AH = 000001xxb (which indicates support for Write functions).

Calling Parameters:

AH = 8BH Add XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory which is to have a directory entry added to it.

DS:SI = pointer to a new XIP_dir entry

Contains a far pointer to a 32 byte area of RAM memory which contains the information of a new XIP application to be added to the XIP directory page (page zero) within the XIP partition. Refer to the structure defined in Table 6-2 of this section. The XIP_STATUS field bits 0-2 of this structure are ignored, but all other fields must be set by the caller.

Results:

These results are valid only if the status returned is PASSED.

DX = XIP_app_handle

The value of the handle assigned to the newly added XIP application.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = 85H	There aren't enough handles to satisfy the application's request. The XIP directory entry was not added.
AH = 87H	There aren't enough logical pages in the XIP partition to satisfy the application's request. The XIP directory entry was not added.
AH = 88H	There aren't enough unallocated logical pages in the XIP partition to satisfy the application's request. The XIP directory entry was not added.
AH = A1H	An XIP application with this name already exists. The XIP directory entry was not added. If your application wants to replace an old XIP application, it must first delete the old one and then add the new one.
AH = A2H	The XIP application name is invalid. The XIP directory entry was not added.
AH = F7H	This function is supported on the media the XIP partition is implemented on. However, the driver or system does not support this function for this media type.
AH = F8H	The specified partition does not exist.
AH = F9H	The XIP directory has insufficient remaining space to add a new entry.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FEH	This function is not supported on the type of media the XIP partition is implemented on (ROM). The XIP directory entry was not added.
AH = FFH	The function failed. Cause unknown.

6.6.13 Copy XIP Page (Both) (Write)

Purpose:

This function allows an XIP utility to copy a new XIP application code and data located in a RAM buffer into the logical pages allocated to a newly added XIP directory entry. The XIP driver does the copying because the type of programmable memory the XIP application is being copied into may not respond to simple memory writes issued by an application program. Not all media will support this operation. Specifically, only programmable devices allow the creation, copying, and deletion of XIP applications. This function is only supported by a driver that returns from Get XIP Version with AH = 000001xxb (which indicates support for Write functions).

Updates to existing memory pages can be made if the card technology and system supports write access.

Calling Parameters:

AH = 8CH	Copy XIP Page function.

AL = partition ID

Contains the ID of the partition which is to have a page copied into it. This must be the same partition that was specified in the Add XIP Directory Entry call used to create the application directory entry.

BX = logical_page_number

The logical page number associated with the XIP application specified by the handle into which the RAM buffer is to be written. If logical page zero is specified, the buffer is copied to the offset in the XIP page specified in the XIP_OFFSET field of the directory entry for this application.

CX = write byte count

Number of bytes to write to XIP page.

DX = XIP_app_handle

The value of the handle assigned to the newly added XIP application. This is the handle value returned by the "Add XIP Directory Entry" function.

DS:SI = pointer to RAM buffer to be copied into logical_page

Contains a far pointer to an area of RAM memory which contains the XIP application code or data to be copied into the logical page specified.

Registers Modified:

AX

Results:

None.

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = 83H	The XIP driver could not find the specified XIP handle. The XIP driver doesn't have any information pertaining to the specified XIP handle. The program has probably corrupted its XIP handle. The buffer was not copied.
AH = 8AH	The logical page to be copied to is out of the range of logical pages allocated to the XIP handle. The buffer was not copied.
AH = F7H	This function is supported on the media the XIP partition is implemented on. However, the driver or system does not support this function for this media type.
AH = F8H	The specified partition does not exist.
AH = FAH	The number of bytes requested to write is larger than the available space in the page.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.

AH = FDH The function failed due to a error in copying. The media supports this function but the logical page the new XIP application is being copied to is not blank and the data cannot be correctly copied. The buffer was not copied.

AH = FEH This function is not supported on the type of media the XIP partition is implemented on (ROM). The buffer was not copied.

AH = FFH The function failed. Cause unknown.

6.6.14 Delete XIP Directory Entry (Both) (Write)

Purpose:

This function allows an XIP utility to delete an old XIP application from the XIP directory. The utility would use this function to name the XIP application to be deleted from the XIP partition. Not all media will support this operation. Specifically, only programmable devices allow the creation, copying, and deletion of XIP applications. This function is only supported by a driver that returns from Get XIP Version with AH = 000001xxb (which indicates support for Write functions).

Calling Parameters:

AH = 8DH Delete XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory which is to have a directory entry deleted from it.

DS:SI = pointer to a old XIP_dir entry to delete

Contains a far pointer to an area of RAM memory which contains the name of an old XIP application to be deleted from the XIP directory page (absolute page zero) within the XIP partition. The old XIP application's directory structure consists of the following elements:

```
XIP_dir_struct          STRUC
  name                  DB    8  DUP  (?)
  ext                   DB    3  DUP  (?)
XIP_dir_struct          ENDS
```

.name

The first member is an 8 byte array which contains the name, padded with blanks (20H), of the old XIP application to be deleted.

.ext

The second member is a 3 byte array which contains the name extension, padded with blanks (20H), of the old XIP application to be deleted.

Results:

None.

Registers Modified:

AX

Status:

CARRY CLEAR PASSED

CARRY SET FAILED

AH = A0H No corresponding handle could be found for the XIP application specified. The XIP directory entry was not deleted.

AH = A1H The XIP application name is invalid. The XIP directory entry was not deleted.

AH = F7H This function is supported on the media the XIP partition is implemented on. However, the driver or system does not support this function for this media type.

AH = F8H The specified partition does not exist.

AH = FCH The card containing the XIP partition has changed since the last XIP API call.

AH = FEH This function is not supported on the type of media the XIP partition is implemented on (ROM). The XIP directory entry was not deleted.

AH = FFH The function failed. Cause unknown.

6.6.15 Erase XIP Partition (Both) (Write)

Purpose:

This function allows an XIP utility to erase the entire contents of an XIP partition in preparation for reuse, The entire XIP partition must be aligned on erase block or device boundaries for this function to only erase the XIP partition and nothing else. This function is only supported by a driver that returns from Get XIP Version with AH = 000001xxb (which indicates support for Write functions).

Calling Parameters:

AH = 8EH Erase XIP Partition function.

AL = partition ID

Contains the ID of the partition to be erased.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = F7H	This function is supported on the media that the XIP partition is implemented on. However, the driver or slot does not support this function for this media type.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP partition has changed since the last XIP API call.
AH = FEH	This function is not supported on the type of media the XIP partition is implemented on (ROM). The XIP partition was not erased.
AH = FFH	The function failed. Cause unknown.

6.6.16 Close XIP Directory Entry (Both) (Write)

Purpose:

This function closes the process of adding a new XIP entry to an XIP partition. Until this function is executed, a newly added XIP application cannot be loaded or executed by an XIP loader. The function "activates" the newly added XIP application by setting the status bits of a newly added XIP directory entry to a value indicating that it may be loaded and executed. Not all media will support this operation. Specifically, only programmable devices allow the creation, copying, and deletion of XIP applications. This function is only supported by a driver that returns from Get XIP Version with AH = 000001xxb (which indicates support for Write functions).

Calling Parameters:

AH = 8FH	Close XIP Directory Entry function.

AL = partition ID

Contains the ID of the partition containing the XIP directory entry which is to be closed.

DX = XIP_app_handle

The value of the handle assigned to the newly added XIP application. This is the handle value returned by the "Add XIP Directory Entry" function.

Registers Modified:

AX

Status:

CARRY CLEAR PASSED

CARRY SET FAILED

AH = 83H The XIP driver could not find the specified XIP handle. The XIP driver doesn't have any information pertaining to the specified XIP handle. The program has probably corrupted its XIP handle. The XIP application directory entry was not closed.

AH = F7H This function is supported on the media the XIP partition is implemented on. However, the driver or slot does not support this function for this media type.

AH = F8H The specified partition does not exist.

AH = FCH The card containing the XIP partition has changed since the last XIP API call.

AH = FEH This function is not supported on the type of media the XIP partition is implemented on (ROM). The XIP application directory entry was not closed.

AH = FFH The function failed. Cause unknown.

6.6.17 Map Extended Segment (EXIP)

Purpose:

This function maps PCMCIA memory cards into or out of the system's extended address space. The specified application on the PCMCIA memory card will appear at the address range returned by this function. A system may support simultaneously mapping of multiple segments.

Calling Parameters:

AH = 90H Contains the Map Extended Segment function code.

AL = partition ID

DX = XIP application handle.

Results:

EDX = System memory address corresponding to the specified application.

ECX = Number of bytes mapped.

The number returned can be more than the number requested.

Note: More memory than was requested can be mapped depending on the granularity of mapping supported by a particular system.

Registers Modified:

AX, ECX, EDX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = F4H	Unable to map requested segment. All hardware mapping currently in use.
AH = F5H	Function not available. Hardware extended memory mapping is not available.
AH = F6H	Function not available. Processor does not support extended memory addressing.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP application has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.18 Unmap Extended Segment (EXIP)

Purpose:

This function maps PCMCIA memory cards out of the system's extended address space. The specified application on the PCMCIA memory card will no longer appear in the address space.

Calling Parameters:

AH = 91H Contains the Unmap Extended Segment function code.

AL = partition ID

DX = XIP application handle.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = 83H	XIP handle not found.
AH = F5H	Function not available. Hardware extended memory mapping is not available.
AH = F6H	Function not available. Processor does not support extended memory addressing.
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP application has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.19 Get Partition ID from Address (Both)

Purpose:

This function returns the ID of the partition onto which the specified system address is currently mapped. If the system address is not currently mapped, failure is indicated.

Calling Parameters:

AH = 92H	Contains the Get Partition ID from Address function code.

ES:DI = 32 bit system address, (may be real, protected, or virtual mode).

Results:

AL = Partition ID

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = F3H	System address not currently mapped.
AH = FCH	The card containing the XIP application has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.20 Get Slot Number (Both)

Purpose:

This function returns the number of the slot that contains the specified partition. If the partition ID is not valid, a failure is indicated.

Calling Parameters:

AH = 93H	Contains the Get Slot Number function code.

AL = partition ID

Results:

AL = slot number.

The specific value returned is vendor specific.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = F8H	The specified partition does not exist.
AH = FCH	The card containing the XIP application has changed since the last XIP API call.
AH = FFH	The function failed. Cause unknown.

6.6.21 Disable Partition ID (Both)

Purpose:

This function marks a partition ID as invalid and allows it to be reassigned to some different XIP partition (via Get XIP Partition IDs).

Calling Parameters:

AH = 94H	Contains the Disable Partition ID function code.
AL = partition ID	

Results:

None.

Registers Modified:

AX

Status:

CARRY CLEAR	PASSED
CARRY SET	FAILED
AH = F8H	The specified partition does not exist.
AH = FFH	The function failed. Cause unknown.

6.7 Example of XIP API Use

An example is included to illustrate some of the more complex processes involved in using an XIP driver and the companion XIP directory structure managed by the driver.

Suppose one wants to install a new XIP application named "WORDPROC.XIP" to an XIP partition within a users PC Memory Card. Further assume that the new XIP application is 109 Kbytes long. Assume that the XIP partition on this hypothetical PC Memory Card already has some XIP applications in it, but absolute-page 17 through absolute-page n within this partition are free. The following steps illustrate how one might copy this new XIP application into the partition. This example assumes an XIP installation tool that adds applications on page boundaries.

1. An XIP-copy-utility would search the existing XIP directory structure, using the "Search for XIP Directory Entry" function, for an existing XIP application with the same name.

2. If an XIP application with the same name already existed in the XIP directory, the XIP-copy-utility might inform the user of this condition and, if instructed to do so by the user, delete the old XIP application by using the "Delete XIP Directory Entry" function.

3. The XIP-copy-utility, knowing the size of the new XIP application, and that no XIP application is in the current XIP directory with the same name as the new one being added, would create for the new application using the "Add XIP Directory Entry" function.

   The data structure for adding this XIP application would look like:

```
XIP_dir_struct    STRUC
  name            DB     "WORDPROC" ;
  ext             DB     "XIP"      ; if necessary
  status          DB     8h         ; this is an EXIP application
  begin           DW     xxxxh      ; entry point offset
  offset          DW     0          ; beginning of page
  reserved        DW     0,0,0      ; reserved words
  creation_time   DW     xxxxh      ; DOS formatting of time bits
  creation_date   DW     xxxxh      ; DOS formatting of date bits
  first_page      DW     xxxxh      ; based on previous entry
  size            DD     (109 *1024); size in Kbytes
XIP_dir_struct    ENDS
```

   After this structure is written into the XIP partition, pages 17, 18, 19...23 are now used by the XIP application named "WORDPROC.XIP".

4. The XIP-copy-utility would then copy the first 16-Kbyte portion of the new XIP application, from whatever media it was contained on, into a RAM buffer . The XIP-copy-utility then copies this buffer into the first logical page allocated to the new XIP application by using the "Copy XIP Page" function. Realize that the type of memory that XIP applications are stored will typically not be normal RAM. It is necessary to have the XIP driver do the copying because it is aware of the nature of the memory on the card. The call may even fail if the memory type happens to be ROM, which is not writeable. The call may also fail if a defect is discovered in the memory in the XIP partition. The important point to remember is that the status of every operation must always be checked.

5. The process begun in step 4 is repeated for logical pages 1, 2, 3, 4, 5, and 6.

6. The last step required is to use the "Close XIP Directory Entry" function. This essentially makes the XIP directory entry and its corresponding application "active" so that it can be loaded and executed.

7. Once all logical pages of the new XIP application have been initialized and the entry has been closed, the process of adding a new XIP application to the partition is complete. At this point, the XIP-copy-utility is done and the user would have a new XIP application in their XIP partition.

6.8 Summary of XIP Status Codes

The following status codes are returned by a comforming XIP driver. Their value, status of the operation, and a brief description of the possible cause are listed in Table 6-6.

Table 6-6: XIP Status Codes

Code	Possible Cause
83H	XIP handle not found.
85H	Insufficient XIP handles to complete the operation.
87H	Insufficient total logical pages within the XIP partition to complete the operation.
88H	Insufficient unallocated logical pages within the XIP directory to complete the operation.
8AH	Logical XIP page out of the range of logical pages allocated to the XIP application.
8BH	Mappable segment address is not mappable by the XIP driver.
A0H	XIP application not found in the XIP directory, or there are no more XIP applications present in this XIP directory.
A1H	XIP application name already exists.
A2H	XIP application name is invalid.
A3H	XIP mapping array contents are invalid.
F3H	Address not currently mapped.
F4H	All mapping hardware currently in use.
F5H	Function not available. Hardware extended memory mapping not available.
F6H	Function not available. Processor does not support extended memory addressing.
F7H	The function (write/erase) is supported on the media the XIP partition is implemented on. However, the driver or system does not support this function for this media type.
F8H	The specified partition does not exist.
F9H	Insufficient space in the XIP directory to complete the operation.
FAH	The number of bytes requested is too large to write.
FBH	Page already mapped into system.
FCH	The card containing the XIP application has changed since the last XIP API call.
FDH	An error occurred in copying the a new XIP application into the XIP partition.
FEH	This function is not supported on the type of media the XIP partition is implemented on (ROM).
FFH	The function failed. Cause unknown.

6.9 Summary of XIP Function Codes

The following function codes are used by a comforming XIP driver. Their value, and the functions they perform are listed in Table 6-7.

Table 6-7: XIP Function Codes

Code	Function
80H	Get XIP Version (Both)
81H	Get XIP Mappable Segments (LXIP)
82H	Get XIP Partition IDs (Both)
83h	Get XIP Handle Range (Both)
84H	Map/Unmap an XIP Handle's Pages (LXIP)
85H	Get XIP Mapping Context Size (Both)
86H	Get XIP Mapping Context (Both)
87H	Set XIP Mapping Context (Both)
88H	Search for XIP Directory Entry (Both)
89H	Get First XIP Directory Entry (Both)
8AH	Get Next Directory Entry (Both)
8BH	Add XIP Directory Entry (Both) (Write)
8CH	Copy XIP Page (Both) (Write)
8DH	Delete XIP Directory Entry (Both) (Write)
8EH	Erase XIP Partition (Both) (Write)
8FH	Close XIP Directory Entry (Both) (Write)
90H	Map Extended Segment (EXIP)
91H	Unmap Extended Segment (EXIP)
92H	Get Partition ID from Address(Both)
93H	Get Slot Number (Both)
94H	Disable Partition ID (Both)

APPENDIX - A

GLOSSARY

SPECIFICATION 2.0

GLOSSARY

A.1 Metaformat Glossary

attribute memory: PCMCIA/JEIDA standard memory cards provide a separate memory address space for recording fundamental card information. This memory is intended to be used by the card manufacturer to record basic configuration information. This memory is selected by asserting the /REG line on the card interface. It is typically, but not necessarily, read-only.

Attribute memory space need not be physically distinct from common memory space; but it must be logically distinct.

basic compatibility layer: The layer of this standard (layer 1) which mandates the use of a card-information structure (CIS) at the beginning of any complying card.

big-endian byte order: A means of specifying the order in which multi-byte numeric objects are recorded, when broken into bytes. Big-endian byte order specifies that the most-significant byte shall be recorded in the lowest byte address; bytes of decreasing significance shall be recorded sequentially in subsequent bytes. Cf. little-endian byte order.

block: For disk-like data formats, a block is the fixed-length sequence of bytes. In such formats, data must usually be read or written as a series of one or more blocks.

byte: In this standard, a byte is eight bits.

byte mapping: The sequence in which byte data is recorded on cards. For 8-bit memory cards, the byte mapping is one-to-one, and not at issue for standardization. For 16-bit and wider cards, the byte mapping within words of the card is arbitrary, and so is governed by this standard.

buffer page: A region of memory on a card used to improve reliability when updating a card. A buffer page typically includes an indication of the region of the card being updated, an image of the desired value for the region of the card, and a flag that indicates that the buffer page is valid. If power fails while a card is being updated, the buffer page can be used to automatically complete the transfer when power is restored.

Card Information Structure: A data structure written at the beginning of every card that complies with this standard, containing information about the formatting and organization of the data on the card.

checksum: An arithmetic error-checking code for data recording based on summing the bytes of data to be checked. Checksums are frequently used by systems that perform error-checking in software.

CIS: Card information structure.

common memory: PCMCIA/JEIDA standard cards provide two memory address spaces. The term "common memory" denotes the primary address space, containing the memory used for application data storage. See also attribute memory.

CRC: Cyclical redundancy check.

cyclical redundancy check: An error-checking code for data recording based on bitwise polynomial division of the data bytes to be checked. As used in this standard, refers to the 16-bit SDLC version of this code, using the polynomial x16 + x12 + x5 + 1, with the check-register initialized to all ones. CRCs are typically used by systems that perform error-checking in hardware.

cylinder: A unit of disk organization. A disk is typically viewed as a collection of cylinders. Each cylinder on a disk is divided into tracks; each track is further divided into sectors. Typically, all of the sectors within a cylinder can be accessed without moving the arm of the disk. See sector.

data organization: The logical organization of data on a card, independent of the data-recording format. The data organization of a memory card will almost always be some kind of file system.

data organization layer: The layer of this standard covering the data organization of the card.

data-recording format: The organization of a memory card into sequences of bytes that are updated or accessed by a single logical operation. The data-recording format of a card includes such details as whether the card's data is organized into blocks of bytes; whether the card includes error checking codes for each block; and so forth. The data-recording format does not specify whether a file system is used. The data-recording format of a card is akin to the physical format of a diskette. Cf. data organization.

data recording format layer: The layer of this standard (layer 2) that specifies the data-recording format of a card.

DOS: The disk operating system for 80x86 architecture systems, such as the IBM PC. DOS is available in several different versions, which are largely compatible with each other; the term generically designates all of them.

EDC: Error-detection code

EEPROM: Electrically-Erasable Programmable Read-Only Memory. A non-volatile memory device which can be programmed electrically, and in which individual bytes can be erased electrically. Usually writes and erasures are much slower than reads.

EPROM: Erasable Programmable Read-Only Memory. A memory device which can be programmed electrically, and erased in bulk by some means, usually by exposure to ultraviolet light.

error-detection code: A numeric code derived from the contents of a data block, used to determine whether the data read from the block are probably correct.

file system: .... An operating-system specified method of structuring data on a mass-storage device. A file system standard consists of a set of data structures and the rules by which those structures are interpreted. We sometimes say that a card has a file system recorded on it; by this we mean that an operating system utility program has placed the appropriate information on the card, allowing the card to be interpreted and manipulated by the operating system.

Not all cards have file systems on them. Some cards are managed directly by application programs.

Flash EPROM: .... A type of EPROM that can be electrically erased. It differs from EEPROM in that generally the entire memory must be erased at once.

FlaSh: .... A trademark of Microsoft, describing a file system designed for use with UV-erasable or Flash EPROM memory cards.

Kbyte: .... kilobyte. 1 Kbyte = 1024 bytes.

little-endian byte order: .... A means of specifying the order in which multi-byte numeric objects are recorded, when broken into bytes. Little-endian byte order specifies that the least-significant byte shall be recorded in the lowest byte address; bytes of increasing significance shall be recorded sequentially in subsequent bytes. Cf. big-endian byte order.

ISO 646 IRV: .... International Standards Organisation standard number 646 (Character codes), Inteernational Reference version. A character set very similar to ASCII, used internationally for representing textual information. It differs from ASCII only in that code 24h represents the international currency symbol rather than the dollar sign ("$"). Except in the alternate/national string tuple, all character data shall be represented using the printing characters from this character set.

LSB: .... Least-significant byte.

metaformat: .... In this standard, the word metaformat is used to encompass the contents, layout, and interpretation of the card information structure. The PCMCIA Metaformat Standard is outlined in Section 5 of this document.

one-time programmable: .... A term describing memory that can be programmed to a specific value once, and thereafter cannot be changed (or can only be revised in a limited way). One-time programmable EPROMs are ordinary EPROMs that have been packaged in such a way that ultra-violet light cannot be used to erase the contents of the EPROM. Such packaging is ususlly less expensive.

OTP: .... One-time programmable

paragraph: .... On Intel 80x86 family machines, a paragraph is a block of sixteen bytes, aligned on a sixteen-byte boundary.

partition: A region of a mass storage device. In this standard, partitions are used to allow a single card to contain two different kinds of data; for example, a card might contain a normal DOS file system in one partition, and directly-executable ROM images in another partition. Most RAM cards will contain only a single partition that contains all the usable storage of the device.

partition check code: A simple method of verifying the contents of an entire partition. A checksum is computed by summing together all the data bytes of the partition; this sum is compared to a value stored in the format tuple that defines the partition. This method is typically used for partitions that change relatively infrequently, such as data partitions in OTP memory.

PCC: Partition check code

PSP: Program-segment prefix. Under DOS, the PSP is the primary data structure for a process, containing its command line, information about exception handling, and so forth.

Release 2: Refers to all *PCMCIA PC Card Standard,* releases 2.0 through current.

reserved: As used in this standard, a reserved field or code value is set aside for use in future standardization. Vendors shall not use reserved fields or code values for any purpose except compliance with future versions of this standard.

sector: As used in this standard, a sector is the fundamental data storage unit of a disk. A sector is the smallest unit of data that can be individually read or updated. Disk sectors correspond to memory card blocks.

TPL: Abbreviation used in symbolic codes to represent the word "tuple".

TSR: Acronym for terminate-and-stay-resident. Under DOS, a TSR is a program that is loaded semi-permanently into memory, extending the system's functionality.

tuple: In this standard, a tuple is a block that appears in the Card Information Structure. Tuples are used to record various items of information about the card layout. All tuples have a common format, shown in Table 19, page 49.

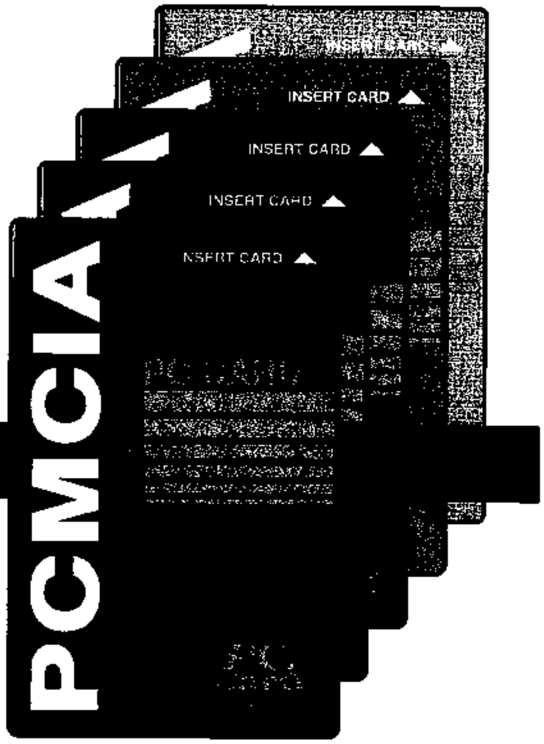
vendor specific: In this standard, this term indicates bits, fields, or code values that are specific to a particular vendor and are not defined by this standard. This standard further distinguishes two kinds of vendor: the card manufacturer, and the supplier of the card data contents.

word: As used in this standard, a word is the smallest addressable unit of a given card. Eight-bit cards have eight-bit words composed of one byte; 16-bit cards have 16-bit words, composed of two bytes; and so forth.

INDEX

PERSONAL
COMPUTER
MEMORY CARD
INTERNATIONAL
ASSOCIATION

PCMCIA STANDARDS



SECTION - 1

INTRODUCTION

INTRODUCTION

1.1 Purpose

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

1.2 Scope

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

1.3 Related Documents

PCMCIA, PC *Card Services Interface Specification*, Release *2.1* ~~2.0~~, *April 1993* ~~November 1992~~, Personal Computer Memory Card International Association.

PCMCIA, *PC Card Standard*, Release *2.1* ~~2.01~~, *April 1993* ~~November 1992~~, Personal Computer Memory Card International Association.

1.4 Terms and Abbreviations

This section describes the terms and abbreviations used in this document:

Term — **Definition**

Adapter .... The hardware which connects a host computer bus to 68-pin PC Card sockets.

ASCIIZ .... A text string in ASCII format terminated with a byte of zero.

CIS .... Acronym for Card Information Structure.

Client .... Within this document, the term client refers to the user of Socket Services, typically Card Services, and not the end-user of the host computer.

End-user .... A person who uses a computer.

Host .... A computer which contains an adapter with PC Card sockets.

Page .... A subdivision of a window. If there is more than one page in a window, all pages are 16 KBytes in size.

PC Card .... A memory or I/O card compatible with the PCMCIA PC Card Standard.

PCMCIA .... Acronym for Personal Computer Memory Card International Association.

Reset .... Refers to the state of a bit within a register. Reset is equivalent to off or zero (0).

Set .... Refers to the state of a bit within a register. Set is equivalent to on or one (1).

Socket... The 68-pin socket a PC Card is inserted in.

Window... An area in a host computer's memory or I/O port space through which a PC Card may be addressed.

XIP... Acronym for eXecute-In-Place. Refers to specification for directly executing code from a PC Card.

SECTION - 2

OVERVIEW

OVERVIEW

Socket Services is the lowest layer in a multi-layer architecture that manages resources on PCMCIA-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 hardware-mapped windows into system memory for both of the device drivers involved in accessing those partitions. Card Services is responsible for handling overlapping requests, insuring that the appropriate partition is mapped into system memory 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 computer's bus to PC Card sockets. Host computers 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 and allow data to be read and/or written on cards which are not mapped into system memory.

For performance reasons, it is often beneficial to map PC Cards into system memory or I/O space. (XIP requires the ability to map PC Card memory arrays into system memory space.) Adapters may or may not provide this capability. An area of PC Card memory and/or I/O space is mapped into the corresponding system area through a window. A memory window is composed of one or more pages. If there are multiple pages in a memory window, all pages within the window are the same size (16 KBytes) and are located contiguously within the window.

Memory windows may address common or attribute memory. Memory windows may overlap in system memory space only if they are time-multiplexed. In other words, only one window at a time may be mapped into system memory at the overlapping addresses. Other windows which also use those addresses must be specifically disabled.

Windows that deal with I/O space do not have any concept of pages. I/O windows respond to I/O bus requests within their range by asserting Card Enable for the socket. It is then up to the PC Card to decode the address lines to determine how to respond to the request. I/O windows may overlap, sharing I/O space, if the socket supports the INPACK signal from the PC Card. However, only one card may respond to an I/O request.

SECTION - 3

FUNCTIONAL DESCRIPTION

FUNCTIONAL DESCRIPTION

3.1 System Architecture

Socket Services is a software interface to the hardware used to manage PC Card sockets in a host computer. 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 attribute memory space to be read, but it does not interpret the Card Information Structure (CIS) that may reside there.

Socket Services is invoked in a processor and operating system dependent manner. All function arguments are passed to Socket Services in a binding specific fashion. Status of the Socket Services request is returned in the status argument. See Appendices for specific binding information. Using functional notation, a Socket Services request generically can be considered as:

```
status = Function(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 IBM-PC compatible platform a ROM BIOS Socket Services interface is handled through Interrupt 1AH with functions based at 80H. A client simply sets the host processor's registers for the function desired and executes the Socket Services software interrupt. Status is returned using the Carry flag ([CF]) and registers specific to the function 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 function requests. It is expected the client of Socket Services will check the Socket Services version to determine the level of service available.

3.3 Configuration

The next step is to enumerate the capabilities of the implementation. This entails determining the number of adapters installed, how many windows and sockets 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 services. 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 the following: card removal or insertion, battery low or dead, and ready/busy 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** function 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** function to determine current PC Card and socket state. This state is used to determine what action should be taken by the client. The *PCMCIA Card Services Interface Specification* describes status change interrupt handling by Card Services.

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** function.

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** function.

SECTION - 4

ASSUMPTIONS AND CONSTRAINTS

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 insure 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 a 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** function. 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** function.

A particular implementation may choose not to provide any mapping of 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** function 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' functions prior to setting new state with the various 'Set' functions.

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 implementation 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 functions. 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 function 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 function 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** function, Socket Services is not intended to be re-entrant. Attempting any other Socket Services function 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 implementations should strive to minimize the amount of time interrupts are disabled. However, a Socket Services implementation 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.

SECTION - 5

PROGRAM INTERFACE

PROGRAM INTERFACE

5.1 Presence Detection

The presence of Socket Services is determined by performing the **GetAdapterCount** request described on page 5-14 . If this function returns with a RETCODE other than SUCCESS, the client may assume that the Socket Services functions 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 Overview of Functions

5.2.1 Non-specific Function

There is one socket services function which applies to the interface in general and not to any objects manipulated by the interface. It is:

GetAdapterCount

5.2.2 Adapter Functions

The following Socket Services functions deal with adapters:

AcknowledgeInterrupt
GetSetPriorHandler
GetSetSSAddr
GetAccessOffsets
GetAdapter
GetSSInfo
GetVendorInfo
InquireAdapter
SetAdapter
VendorSpecific

5.2.3 Socket Functions

The following Socket Services deal with sockets:

GetSocket
GetStatus
InquireSocket
ResetSocket
SetSocket

5.2.4 Window Functions

The following Socket Services functions deal with windows:

GetPage
GetWindow
InquireWindow
SetPage
SetWindow

WARNING:

Windows 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.

5.2.5 Error Detection and Correction Functions

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

GetEDC
InquireEDC
PauseEDC
ReadEDC
ResumeEDC
SetEDC
StartEDC
StopEDC

5.2.6 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.

5.2.7 Reserved Functions

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

5.3 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 Socket Services function 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 in a host systems of a window used to map a PC Card into a host's memory or I/O spce
BCD	Binary Coded Deximal value. For example, 0221H represents 2.21.
BYTE	An 8-bit quantity.
COUNT	Number of objects of the specified type.
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 8 significant bits.
FLAGS16	Bit-mapped field with 16 significant bits.
FLAGS32	Bit-mapped field with 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 a PC Card's attribute or common memory plane.
PAGE	Subdivision of a window. Ranges from zero to one less than the number of pages in a window. Windows are 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 PWRENTRY 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 function 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**.

SPEED.. Encoded value representing a memory window access speed. Format defined by PC Card Standard Release 2.01.

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.

5.4 Function Descriptions

Each Socket Services function 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.

Function 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 function. If the verb is Get the current configuration of the item is returned. If the verb is Set the item is configured by the function.

The following notation conventions are used:

Convention	Meaning
bold	Bold type is used for keywords. For example, the names of functions, 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.

5.4.1 AcknowledgeInterrupt

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

The **AcknowledgeInterrupt** function 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 computer.
Sockets	O	Returns a bit-map representing the sockets which have experienced a status change.

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** function. 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** function. 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** function.

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 function 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 function 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:***
>
> ***Acknowledge Interrupt takes place within the status change hardware interrupt. Socket Services must not enable interrupts at any time during the processing of an Anknowledge Interrupt request.***

See Also **GetStatus**

5.4.2 GetAccessOffsets

```
RETCODE GetAccessOffsets (Adapter, Mode, NumDesired, pBuffer, NumAvail)
        ADAPTER   Adapter;
        BYTE      Mode;
        COUNT     NumDesired;
        PTR       pBuffer;
        COUNT     NumAvail;
```

The **GetAccessOffsets** function 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. Adapters which access PC Card memory through windows mapped into host system memory do not support this function.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host computer.
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_FUNCTION	if request is not supported
BAD_MODE	if Mode is not supported

Comments

It is assumed that 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 an internal table 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

The actual arguments passed to the above access routines for an Intel x86 processor are defined in Appendix F of the PCMCIA Card Services Interface Specification.

See Also GetSetSSAddr

5.4.3 GetAdapter

RETCODE GetAdapter *(Adapter, State, SCRouting)*

ADAPTER	*Adapter;*
FLAGS8	*State;*
IRQ	*SCRouting;*

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

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host computer
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 Set Adapter function. If reset, adapter hardware is fully powered and fully functional.
AS_MAINTAIN	If set, all adapter and socket configuration information is being 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.

Parameter	I/O	Description
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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with this function, make changes and then use the **SetAdapter** function to modify the configuration without having to create initial values for each parameter.

See Also **InquireAdapter, SetAdapter**

5.4.4 GetAdapterCount

```
RETCODE GetAdapterCount (TotalAdapters, Signature)
        COUNT       TotalAdapters;
        SIGNATURE   Signature;
```

The **GetAdapterCount** function 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.
Signature	O	If 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.

Return Codes:

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.

Comments

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

See Also **GetSSInfo**

5.4.5 GetEDC

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

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

The GetEDC function 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 computer.
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.

Parameter	I/O	Description
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.

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 function. This is intended to allow clients of Socket Services to retrieve current configuration information with this function, make changes and then use **SetEDC** to modify the configuration without having to create initial values for each parameter.

See Also **InquireEDC, SetEDC, Start EDC, PauseEDC, ResumeEDC, StopEDC, ReadEDC**

5.4.6 GetPage

```
RETCODE GetPage (Adapter, Window, Page, State, Offset)
        ADAPTER     Adapter;
        WINDOW      Window;
        PAGE        Page;
        FLAGS8      State;
        OFFSET      Offset;
```

The **GetPage** function 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 computer.
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.

Parameter	I/O	Description
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*~~	*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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with this function, make changes and then use the **SetPage** function 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**

[illegible] GetPage

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

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

The **GetPage** function 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 computer.
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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with this function, make changes and then use the **SetPage** function 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**.

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

5.3.7 GetSetPriorHandler

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

The **GetSetPriorHandler** function 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 computer.
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_FUNCTION	If request is to Set a prior handler for a ROM-based handler which is hard-coded to chain to another type of handler

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 implementation as its prior handler.

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

WARNING:

This function should only be used with the first adapter serviced by a Socket Services handler as returned by the GetSSInfo function. 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:

Clients should not attempt to add Socket Services which increase the number of adapters and/or sockets supported. 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 implementation with an updated version.

5.4.8 GetSetSSAddr

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

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

The **GetSetSSAddr** function 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 computer.
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.
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.
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.

Comments

Some Socket Services 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 implementation needs to address besides the main data segment.

A Card Services using an entry point returned by this function 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 function, 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

Warning:

This function should only be used with the first adapter serviced by a Socket Services handler as returned by the GetSSInfo function. 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_FUNCTION	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**

5.4.9 GetSocket

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

ADAPTER	*Adapter;*
SOCKET	*Socket;*
FLAGS8	*SCIntMask;*
PWRINDEX	*VccLevels;*
PWRINDEX	*VppLevels;*
FLAGS8	*State;*
FLAGS8	*CtlInd;*
IRQ	*IREQRouting;*
FLAGS8	*IFType;*

The **GetSocket** function 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 computer
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**.
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 signals. This is two indicies 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**.
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 setting. Settings are mutually exclusive. This parameter may be set to one of the following values:

IF_MEMORY	If set, socket interface is set to Memory-Only as defined by the PCMCIA PC Card Standard Release 2.01.
IF_IO	If set, socket interface is set to I/O and Memory as defined by the PCMCIA PC Card Standard Release 2.01.

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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with with this function, make changes and then use **SetSocket** to modify the configuration without having to create initial values for each parameter.

See Also **InquireSocket, SetSocket**

5.4.10 GetSSInfo

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

The **GetSSInfo** function 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 computer. 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 PCMCIA Socket Services Interface Specification compliance level as a Binary Coded Decimal (BCD) value. If the handler is compliant with Release 2.0 of the PCMCIA Socket Services Interface Specification, 0200H is returned.
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**

5.4.11 GetStatus

```
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** function 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 computer.
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 SMB_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. 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 RDY/BSY (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 changes. Values reported are always based on the signal levels at the socket. If the IFType is IF_IO, this function does NOT read status from the Pin Replacement Register. This is the responsibility of the client.
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

Comments

This function 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, Get Socket, SetSocket**

5.4.12 GetVendorInfo

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

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

The **GetVendorInfo** function 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 computer.
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 implementor.
pBuffer	I	If *Type* is zero (0), this parameter points to a client-supplied buffer to be filled with an ASCIIZ string describing the implementor. 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.

Parameter	I/O	Description
Release	O	Vendor's release number in BCD format. Each time a vendor releases a new version of their Socket Services implementation, 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 implementation. A subsequent release of an updated version compliant with the same level of the PCMCIA Socket Services Interface Specification should change this value according to the vendor's change control procedures. The first release of an implementation compliant with a new PCMCIA 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_FUNCTION	if *Type* is invalid

5.4.13 GetWindow

```
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 function 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 computer.
Window	I	Specifies a physical window on the adapter.
Socket	O	Returns the physical socket the window is currently assigned.
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	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 PC Card into the host systems's I/O space. If reset, window maps common or attribute memory on a PC Card into the host systems's memory space.
WS_ENABLED	If set, window is enabled and mapping a PC Card into the host system memory or I/O 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 PC 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. This is defined in the *PCMCIA PC Card Standard*, Release 2.01, Section 5.2.7.1.3 Device ID Speed Field. Tables 5-12: Device Speed Codes and 5-13: Extended Device Speed Codes in the PC Card Standard list specific values.

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 undefined 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 system memory or I/O 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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with this function, make changes and then use the **SetWindow** function 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 is managed by **Get Page** and **Set Page** requests.

*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, SetWindow, GetPage, SetPage**

5.4.14 InquireAdapter

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

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

The InquireAdapter function 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 computer
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 5-34.

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 VDC 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 5-35.

NumSockets	O	Returns the number of sockets provided by the adapter.
NumWindows	O	Returns the number of windows provided by the adapter.
NumEDCs	O	Returns the number of Error Detection Code (EDC) generators 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 power levels. 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 12v. It is probably not valid to set Vcc to No Connect and the Vpp pins to 12v at the same time. 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 = {
   24,                     //Size of client-supplied buffer is 24 bytes
   24,                     //Size of data returned is 24 bytes
   {0,                     //Indicators, power and data bus width
                           //controlled at the socket
   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 {  //Same format as Socket characteristics
   FLAGS16              AdpCaps;// except this field has different values
   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 function, 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 to 12v results in 12v being applied to all of the sockets on the adapter. Socket Services does not remove 12v from the Vpp lines until all sockets set Vpp 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.

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 25.5VDC 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

5.4.15 InquireEDC

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

ADAPTER	*Adapter;*
EDC	*EDC;*
SKTBITS	*Sockets;*
FLAGS8	*Caps;*
FLASG8	*Types;*

The **InquireEDC** function 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 computer.
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 bidirectional code generation. If reset, EDC generator does not support bidirectional 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** functions are not available.

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

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 function 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 function for each generator.

Socket Services supports two types of EDC generation: 8-bit checksums and 16-bit CRC SDLC. 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. Bidirectional 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** function.

See Also **GetEDC, SetEDC, Start EDC, PauseEDC, ResumeEDC, StopEDC, ReadEDC**

5.4.16 InquireSocket

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

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

The **InquireSocket** function 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 computer
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. This parameter is a combination of the values described below:

Value	Meaning
SBM_WP	PC Card WP signal (write-protect).
SBM_LOCKED	Externally generated signal 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 signal indicating a request to eject a PC Card from the socket has been made.

SBM_INSERT	Externally generated signal indicating a request to insert a PC Card into the socket has been made.
SBM_BVD1	PC Card BVD1 signal. When set, this signal indicates the battery is no longer serviceable.
SBM_BVD2	PC Card BVD2 signal. When set, this signal indicates the battery is weak.
SBM_RDYBSY	PC Card RDY/BSY signal.
SBM_CD	PC Card Card Detect signals.

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 paramter.

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 signal (write-protect) state.
SBM_LOCKED	Indicator for externally generated signal 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 signal.
SBM_BATT	Indicator for state of BVD1 and BVD2 signals.
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 = {
   10,                   // Size of client-supplied buffer is 10 bytes
   10,                   // Size of data returned is 10 bytes
   ~~{0,~~                   ~~// Indicators, power and data bus width~~
                         ~~//    controlled at the socket~~
   {IF_MEMORY \ IF_10,   // 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
};
```

See Also GetSocket, SetSocket

Socket Characteristics Structure

```
typedef struct tagSCHARTBL {           // Same as adapter characteristics
   FLAGS16            SktCaps;                  // except for this member
   FLAGS32            ActiveHigh;
   FLAGS32            ActiveLow;
} 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 as defined by PCMCIA PC Card Standard Release 2.01.
IF_IO	Socket supports I/O and Memory interface as defined by PCMCIA PC Card Standard Release 2.01.

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 normal PC Card IREQ signals may be shared in a host system.

5.4.17 InquireWindow

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

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

The **InquireWindow** function 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 computer
Window	I	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 function provides two tables of information. The **MEMWINTBL** structure is defined on page 5-42. The **IOWINTBL** structure is defined on page 5-46.

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 programable. 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 PC Card into the host system memory space.
WC_ATTRIBUTE	If set, window may be used to map the attribute memory plane of a PC Card into the host system memory space.
WC_IO	If set, window may be used to map I/O ports on a PC Card into the host system I/O space.
WC_WAIT	If set, window uses the -WAIT signal from a 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**

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 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 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, PC Card offsets are required to be specified to **SetPage** in increments of the size of the window.

If reset, 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 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 insure 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 PC Card memory mapped into host system memory space.

If reset, the window may not be write-protected to prevent writing PC Card memory mapped into host system memory space.

Write-protection is enabled and disabled with the **SetPage** function 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 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 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 as defined in the *PCMCIA PC Card Standard*, Release 2.01, Section 5.2.7.1.3 Device ID Speed Field. Tables 5-12: Device Speed Codes and 5-13: Extended Device Speed Codes. in the PC Card Standard list specific values. For Socket Services, Bit 7 of the *Slowest* and *Fastest* members is reserved and is reset to zero (0).

The Device Speed Code values in Table 5-12 of the PC Card Standard 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 Table 5-13: Extended Device Speed Code.

Memory windows map accesses to host system memory into accesses to memory 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** *function. 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** *function.*

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 system I/O 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 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 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 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 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 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.

5.4.18 PauseEDC

```
RETCODE PauseEDC (Adapter, EDC)
        ADAPTER     Adapter;
        EDC         EDC;
```

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

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

5.4.19 ReadEDC

```
RETCODE ReadEDC (Adapter, EDC, Value)
        ADAPTER     Adapter;
        EDC         EDC;
        WORD        Value;
```

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

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host computer.
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, all 16-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**

[illegible] ResetSocket

RETCODE ResetSocket *(Adapter, Socket)*

```
        ADAPTER     Adapter;
        SOCKET      Socket;
```

The **ResetSocket** function 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 computer
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 function toggles the RESET pin of the card in the specified socket on the specified adapter.

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

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

IFType set to IF_MEMORY

IREQRouting disabled

Vcc, Vpp1 and Vpp2 set to 5VDC

All windows, pages and EDC generators disabled.

5.4.21 ResumeEDC

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

The **ResumeEDC** function pauses 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 computer.
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 function is used to resume EDC generation so some accesses to a PC Card are not involved in the computation of an EDC value. This function is only supported if EC_PAUSABLE is set in the **InquireEDC** *Caps* parameter.

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

[illegible] SetAdapter

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

The SetAdapter function sets the configuration of the specified adapter.

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

Parameter	I/O	Description
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** function. This is intended to allow clients of Socket Services to retrieve current configuration information with **GetAdapter**, make changes and then use this function to modify the configuration without having to create initial values for each parameter.

See Also **InquireAdapter, GetAdapter**

5.4.20 SetEDC

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

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

The SetEDC function 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 computer.
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.

Parameter	I/O	Description
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.

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 function. This is intended to allow clients of Socket Services to retrieve current configuration information with GetEDC, make changes and then use this function to modify the configuration without having to create initial values for each parameter.

See Also **InquireEDC, GetEDC, Start EDC, PauseEDC, ResumeEDC, StopEDC, ReadEDC**

5.4.24 SetPage

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

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

The SetPage function configures 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 computer.
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.

Parameter	I/O	Description
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 function. This is intended to allow clients of Socket Services to retrieve current configuration information with GetPage, make changes and then use this function 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 Inquire*Window*~~Adapter~~.

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**

5.4.25 SetSocket

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

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

The SetSocket function 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 computer
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 interrupts 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.
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.**
VppLevels	I	Sets current power level of Vpp 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**.
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. If IFType is IF_IO, 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_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. Settings are mutually exclusive. This parameter may be set to one of the following values:

IF_MEMORY	If set, socket interface is set to Memory-Only as defined by the PCMCIA PC Card Standard Release 2.01.
IF_IO	If set, socket interface is set to I/O and Memory as defined by the PCMCIA PC Card Standard Release 2.01.

Return Codes:

SUCCESS	if *Adapter* and *Socket* are valid
BAD_ADAPTER	if *Adapter* is invalid
~~BAD_ATTRIBUTE~~	~~if IREQRouting or IFType not supported~~
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 VPP level is invalid*

Comments

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

See Also **InquireSocket, GetSocket**

5.4.26 SetWindow

```
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 function sets the configuration of the window specified by the input parameters.

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host computer.
Window	I	Specifies a physical window on the adapter.
Socket	I	Assigns the window to the specified socket.
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. This parameter can be a combination of the following values:

Value	Meaning
WS_IO	If set, window maps registers on a PC Card into the host systems's I/O space. If reset, window maps common or attribute memory on a PC Card into the host systems's memory space.
WS_ENABLED	If set, window is enabled and maps a PC Card into the host system memory or I/O 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 PC 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 configured for EISA I/O mapping.

If reset, window is configured for 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 are configured to generate card enables.

If set, accesses to I/O ports in EISA common I/O areas are configured to be 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. This is defined in the *PCMCIA PC Card Standard,* Release 2.01, Section 5.2.7.1.3 Device ID Speed Field. Tables 5-12: Device Speed Codes and 5-13: Extended Device Speed Codes in the PC Card Standard list specific values.

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 function. This is intended to allow clients of Socket Services to retrieve current configuration information with GetWindow, make desired changes and then use this function 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 is managed by Get Page and Set Page requests.

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, GetPage, SetPage**

5.23 StartEDC

RETCODE StartEDC *(Adapter, EDC)*

ADAPTER	*Adapter;*
EDC	*EDC;*

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

Parameter	I/O	Description
Adapter	I	Specifies a physical adapter on the host computer.
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 function 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**

5.4.28 StopEDC

RETCODE StopEDC *(Adapter, EDC)*

ADAPTER	*Adapter;*
EDC	*EDC;*

The StopEDC function 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 computer.
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**

5.4.29 VendorSpecific

RETCODE VendorSpecific *(Adapter ...)*
    ADAPTER *Adapter;*

This function is vendor specific. The function is reserved for vendor's 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 computer.

Return Codes:

SUCCESS	if parameters are valid
Other return codes are specific to the Socket Services implementation	

Comments

This function may have additional parameters that are specific to a particular vendor's implementation.

Before using this function, a client should use the Get Vendor Info function to confirm the implementor to validate whether the vendor specific functions are available.

See Also **Get Vendor Info**

SECTION - 6

USING SOCKET SERVICES

USING SOCKET SERVICES

This section describes how various functions 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** function is used for presence detection returning *Signature* and *TotalAdapters*. Next, the client verifies that the proper version of Socket Services is installed. The client checks that a compatible version is installed by verifying the Socket Services Compliance Level returned by **GetSSInfo**. If this version is acceptable, the client may also wish to verify whether the implementor's version is acceptable by checking the ASCIIZ string and Implementation Version Number returned by **GetVendorInfo**. This last step is optional.

Once an acceptable Socket Services implementation 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** function is used to determine a specific window's characteristics. The **InquireSocket** function is used to determine each socket's characteristics.

Error Detection capability is determined in the same manner, using the **InquireEDC** function. 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 enterable. 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 **Acknowledge Interrupt** function to determine which socket experienced the status change.

The **Acknowledge Interrupt** function 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 function. 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 **Get Adapter Count** function and all functions 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.

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 insure 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 inserted cards. These [illegible] are expected to be utilized by card-aware drivers through a higher-level software service.

APPENDIX - A

RETURN CODES

A.1 Return Codes

Return Code	Description
SUCCESS	The request succeeded
BAD_ADAPTER	Specified adapter is invalid
BAD_ATTRIBUTE	Specified attribute is invalid
BAD_BASE	Specified base system memory address is invalid
BAD_ED	Specified EDC generator is invalid
BAD_IRQ	Specified IRQ level is invalid
BAD_OFFSET	Specified PCMCIA card offset is invalid
BAD_PAGE	Specified page is invalid
READ_FAILURE	Unable to complete read request
BAD_SIZE	Specified size is invalid
BAD_SOCKET	Specified socket is invalid
BAD_TYPE	Window or interface type specified is invalid
BAD_VCC	Specified Vcc power level index is invalid
BAD_VPP	Specified Vpp1 or Vpp2 power level index is invalid
BAD_WINDOW	Specified window is invalid
WRITE_FAILURE	Unable to complete write request
NO_CARD	No PC Card in socket
BAD_FUNCTION	Function not supported
BAD_MODE	Requested processor mode not supported
BAD_SPEED	Specified speed is unavailable
BUSY	Unable to process request at this time - retry later

APPENDIX - B

PC-COMPATIBLE BINDING

PC-COMPATIBLE BINDING

B.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 in MS-DOS compatible environments as a device driver or Terminate and Stay Resident (TSR) program.

B.2 Adapters Supported

The Socket Services interface allows multiple adapters containing one or more PC Card sockets. Since the Adapter number 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).

B.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 Inquire Adapter).

B.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.

B.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. 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).

B.6 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 implementation 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.

B.7 Calling Conventions

The interface uses a common set of conventions for all functions. They are described below.

B.7.1 Register Usage

Whenever possible, the interface shall use the 80x86 CPUs registers to pass arguments and return status. Registers are used consistently among the various functions with the following guidelines:

Entry:	[AH]	Function desired
	[AL]	*Adapter*
	[BH]	*Window*
	[BL]	*Page* or *Socket*
	[CX]	*Counts*
	[DX]	*Attributes*
	[DS]:[(E)SI]	Reserved in ROM BIOS Int 1AH Interface
	[ES]:[(E)DI]	Pointer to *Buffer*
	[DI]	*Offset* (4 KByte units)
Exit:	[CF]	Status (set = error, reset = success)
	If [CF] set	
	[AH]	Non-SUCCESS RETCODE
	else	
	[AH]	SUCCESS RETCODE

Please note that these are guidelines used to develop the function interfaces and exceptions have been made for specific functions.

Whenever possible the interface preserves the contents of all registers unless they are used to return information. The only exception is the [AH] register which is used for RETCODE.

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.

B.7.2 Error Reporting

For all functions, the [CF] indicates whether the function was successful. If the [CF] is reset on exit, the function was successful. If the [CF] is set on exit, the function 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 function.

B.8 Individual Function Bindings

The following sections describe how the individual functions are bound to IBM-PC compatible architectures.

B.8.1 AcknowledgeInterrupt

Entry:	[AH]	ACK_INTERRUPT
	[AL]	Adapter (0.. Max Adapter)
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**
	[CX]	*Sockets*

B.8.2 GetAccessOffsets

Entry:	[AH]	ACCESS_OFFSETS	
	[AL]	*Adapter*	
	[BH]	*Mode*	
		00 = Real Mode	02 = 16:32 Protect
		01 = 16:16 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 function 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.

B.8.3 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

B.8.4 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 function 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, un-related handlers. There is no guarantee these other, un-related 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 function to insure the return value is from Socket Services and not just left over in the register from prior client activity.

B.8.5 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

B.8.6 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)

B.8.7 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 Set Prior Handler 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 Get Prior Handler request to this Socket Services with a replacement Socket Services implementation.

B.8.8 GetSetSSAddr

Entry:	[AH]	SS_ADDR	
	[AL]	*Adapter*	
	[BH]	*Mode*	
		00 = Real Mode	02 = 16:32 Protect
		01 = 16:16 Protect	03 = 00:32 Protect
		All other values are reserved and must not be used.	
	[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 function 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 function for any mode other than real, the client must establish a pointer to the main data area in [DS]:[(E)SI].

Note:
Subfunction 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

B.8.9 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*****)**
		~~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~~
	[CH]		*Vcc level*
	[CL]		*VppLevels,* Upper Nibble - Vpp1, Lower Nibble - Vpp2
	[DH]		*State* (Uses same bit masks as *SCIntMask*)
	[DL]		*CtlInd* (Uses same bit masks as InquireSocket)
		~~Bit 4~~	~~SBM_LOCK~~
		~~Bit 5~~	~~SBM_BATT~~
		~~Bit 6~~	~~SBM_BUSY~~
		~~Bit 7~~	~~SBM_XIP~~
	[DI]		High Byte - *IFType, Low Byte - IREQRouting*
		Bit 0 - 4	IRQ level
		Bit 6 ~~5~~	IRQ_HIGH
		Bit 7	IRQ_ENABLED
		Bit 8	IF_MEMORY
		Bit 9	IF_IO

B.8.10 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*

B.8.11 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*)

B.8.12 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*

B.8.13 GetWindow

Entry:	[AH]	GET_WINDOW
	[AL]	*Adapter*
	[BH]	*Window*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**
	[BL]	*Socket*
	[CX]	*Size* (Bytes for I/O windows, 4 KByte units for Memory windows)
	[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*
	[DI]	*Base* (Bytes for I/O windows, 4 KByte units for Memory windows)

B.8.14 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*
	[ES]:[(E)DI]	*pBuffer* with adapter characteristics and power management tables

B.8.15 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]	*Type*
	Bit 0	ET_CHECK8
	Bit 1	ET_SDLC16

B.8.16 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 (same as* SCIntCaps with the following exceptions)
	Bit 4	SBM_LOCK
	Bit 5	SBM_BATT
	Bit 6	SBM_BUSY
	Bit 7	SBM_XIP
	[ES]:[(E)DI]	*pBuffer* with socket characteristics

B.8.17 InquireWindow

Entry:	[AH]	INQ_WINDOW
	[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_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

B.8.18 PauseEDC

Entry:	[AH]	PAUSE_EDC
	[AL]	*Adapter*
	[BH]	*EDC*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.19 ReadEDC

Entry:	[AH]	SET_EDC
	[AL]	*Adapter*
	[BH]	*EDC*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**
	[DX]	*Value*

B.8.20 ResetSocket

Entry:	[AH]	RESET_SOCKET
	[AL]	*Adapter*
	[BL]	*Socket*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.21 ResumeEDC

Entry:	[AH]	RESUME_EDC
	[AL]	*Adapter*
	[BH]	*EDC*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.22 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**

B.8.23 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**

B.8.24 SetPage

Entry:	[AH]	SET_PAGE
	[AL]	*Adapter*
	[BH]	*Window*
	[BL]	*Page*
	[DL]	*State (same as GetPage)*
	[DI]	*Offset* (4 KByte units)
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.25 SetSocket

Entry:	[AH]	SET_SOCKET
	[AL]	*Adapter*
	[BL]	*Socket*
	[BH]	*SCIntMask (same as GetSocket)*
	[CH]	*Vcc level,* Upper Nibble - reserved, Lower nibble - Vcc
	[CL]	*VppLevels,* Upper Nibble - Vpp1, Lower Nibble - Vpp2
	[DH]	*State* (same as GetSocket)
	[DL]	*CtlInd* (same as GetSocket)
	[DI]	High Byte - *IFType, Low Byte - IREQRouting (same as GetSocket)*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.26 SetWindow

Entry:	[AH]	SET_WINDOW
	[AL]	*Adapter*
	[BH]	*Window*
	[BL]	*Socket*
	[CX]	*Size* (Bytes for I/O windows, 4 KByte units for Memory windows)
	[DH]	*State (same as GetWindow)*
	[DL]	Speed
	[DI]	*Base* (Bytes for I/O windows, 4 KByte units for Memory windows)
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.27 StartEDC

Entry:	[AH]	START_EDC
	[AL]	*Adapter*
	[BH]	*EDC*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.28 StopEDC

Entry:	[AH]	STOP_EDC
	[AL]	*Adapter*
	[BH]	*EDC*
Exit:	[CF]	Status, set = error, reset = success
	[AH]	**RETCODE**

B.8.29 VendorSpecific

Entry:	[AH]	VEND_SPECIFIC
	[AL]	*Adapter*

All other registers are vendor specific

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

B.9 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.

; ----- Function 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_INFOEQU         9DH
ACK_INTERRUPT  EQU         9EH
PRIOR_HANDLER  EQU         9FH
SS_ADDR        EQU         0A0H
ACCESS_OFFSETS EQU         0A1H
                                    ;  A2H 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
                                      ;  5 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
                                      ;  CH is reserved for
                                      ;  historical purposes
BAD_TYPE        EQU         0DH~~0CH~~
BAD_VCC         EQU         0EH~~0DH~~
BAD_VPP         EQU         0FH~~0EH~~
                                      ;  ~~FH and~~ 10H ~~are~~ is reserved for
                                      ;  historical purposes
BAD_WINDOW      EQU         11H
WRITE_FAILURE   EQU         12H
                                      ;  13H is reserved for
                                      ;  historical purposes
NO_CARD         EQU         14H
BAD_FUNCTION    EQU         15H
BAD_MODE        EQU         16H
BAD_SPEED       EQU         17H
BUSY            EQU         18H

; ----- Defined data types

ADAPTER         TYPEDEF     BYTE
BASE            TYPEDEF     WORD
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
WSIZE           TYPEDEF     WORD       ;  SIZE is reserved by MASM 6.0
SOCKET          TYPEDEF     BYTE
SPEED           TYPEDEF     BYTE
WINDOW          TYPEDEF     BYTE
SKTBITS         TYPEDEF     WORD

; ----- Structures

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

CHARTBL         STRUCT                 ;  Inquire Adapter and Inquire Socket
                UNION
AdpCaps         FLAGS16     ?
SktCaps         FLAGS16     ?
                ENDS
ActiveHi        FLAGS32     ?
ActiveLo        FLAGS32     ?
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
```

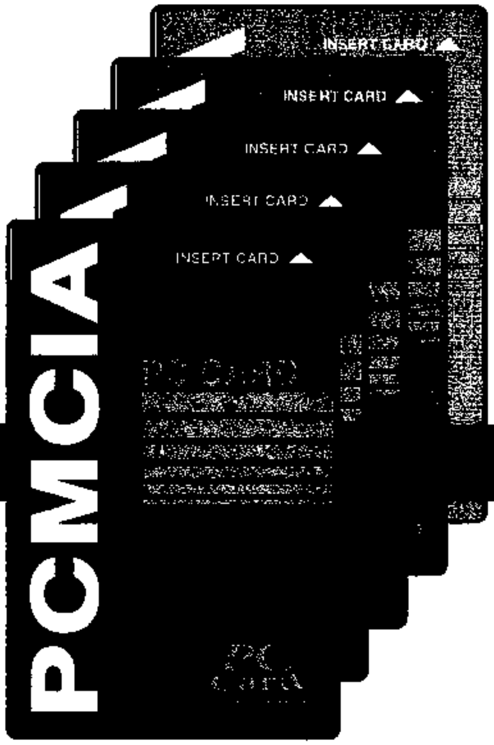
```
EISASlot       FLAGS8      ?        ;  Upper 4 I/O Address lines for EISA
IOWINTBL       ENDS

; ----- 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

; ----- 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

; ----- 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_CALIGN      EQU         0000000010000000B~~

; ----- 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 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

; ----- Interface bit-masks

IF_MEMORY       EQU         00000001B
IF_IO           EQU         00000010B

; ----- 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
```

PERSONAL
COMPUTER
MEMORY CARD
INTERNATIONAL
ASSOCIATION

PCMCIA STANDARDS



SECTION - 1

INTRODUCTION

INTRODUCTION

1.1 Purpose

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

1.2 Scope

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

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

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

1.3 Related Documents

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

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

PCMCIA PC Card Standard, Release *2.1* ~~2.01~~, *April 1993* ~~November 1992~~, Personal Computer Memory Card International Association.

PCMCIA, *PCMCIA Socket Services Interface Specification*, Release *2.1*~~2.0~~, *April 1993* ~~November 1992~~, Personal Computer Memory Card International Association.

1.4 Terms and Abbreviations

Adapter.......................................The hardware which connects a computer bus to PC Card sockets.

ASCIIZ.......................................A text string in ASCII format terminated with a byte of zero.

CIS...Acronym for Card Information Structure.

Client...A user of Card Services functions. May be a device driver, utility program or application program.

EISA..Acronym for Extended Industry Standard Architecture. Refers to an expansion bus promoted by manufacturers of IBM-compatible personal computers that feature 32-bit addressing and bus-mastering capabilities. Not compatible with Micro Channel. Compatible with ISA 8-bit and 16-bit adapter cards.

End-user....................................A person who uses a computer.

Handle.......................................A Card Services assigned identifier associated with Card Services managed system resources. ~~Must be non zero to be valid.~~

Host...The computer which contains an adapter with one or more sockets.

ISA .... Acronym for Industry Standard Architecture. Refers to an IBM-compatible expansion bus of the type incorporated in IBM-AT compatible personal computers. Uses 16-bit addressing.

Micro Channel .... Micro Channel Architecture. Refers to an IBM expansion bus of the type incorporated in some of the personal computers in the PS/2 line. Features 32-bit addressing and bus-mastering capabilities. Not compatible with ISA or EISA.

MTD .... Acronym for Memory Technology Driver. Embedded or installable component of Card Services that contains device-specific read, write, copy and erase algorithms.

Page .... A subdivision of a window. If there is more than one page in a window, all pages are 16 KBytes in size.

Partition .... A portion of a PC Card memory region used for one purpose, e.g. a file system partition.

PC Card .... A memory or I/O card compliant with the PCMCIA standard.

PCMCIA .... Acronym for Personal Computer Memory Card International Association.

Region .... A homogeneous area of memory on a PC Card using one type of memory device.

Memory Handle .... A Card Services assigned identifier for a memory area on a PC Card.

Reset .... Refers to the state of a bit within a register or variable. Reset is equivalent to off or zero(0).

Set .... Refers to the state of a bit within a register or variable. Set is equivalent to on or one(1).

Socket .... The 68-pin PCMCIA socket or connector. This where the PC Card gets plugged in.

User .... Within this document the term user refers to the user of Card Services, typically a higher-level software layer such as a client, and not the end-user of the host computer.

Window .... An area in a host computer's memory or I/O port space where a PC Card is addressed.

XIP .... Acronym for eXecute-In-Place. Refers to specification for directly executing code from a PC Card.

OVERVIEW

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

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

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

This document is divided into several sections:

- Section 3 is a functional description of the Card Services interface. The overall architecture is discussed with special focus on the programming interface, function groupings, callback interfaces, reported events, Memory Technology Drivers, and how Card Services processes the Card Information Structure.
- Section 4 identifies several assumptions and constraints which apply globally to the Card Services interface. The reader should keep these points in mind while reviewing other sections of the specification.
- Section 5 discusses each of the functions provided by Card Services in detail. The purpose of each function is described with its input and output parameters.
- The appendices enumerate the codes used to identify function, event and return values. The function reference for the MTD Helper Functions also appears in the appendix.

SECTION - 3

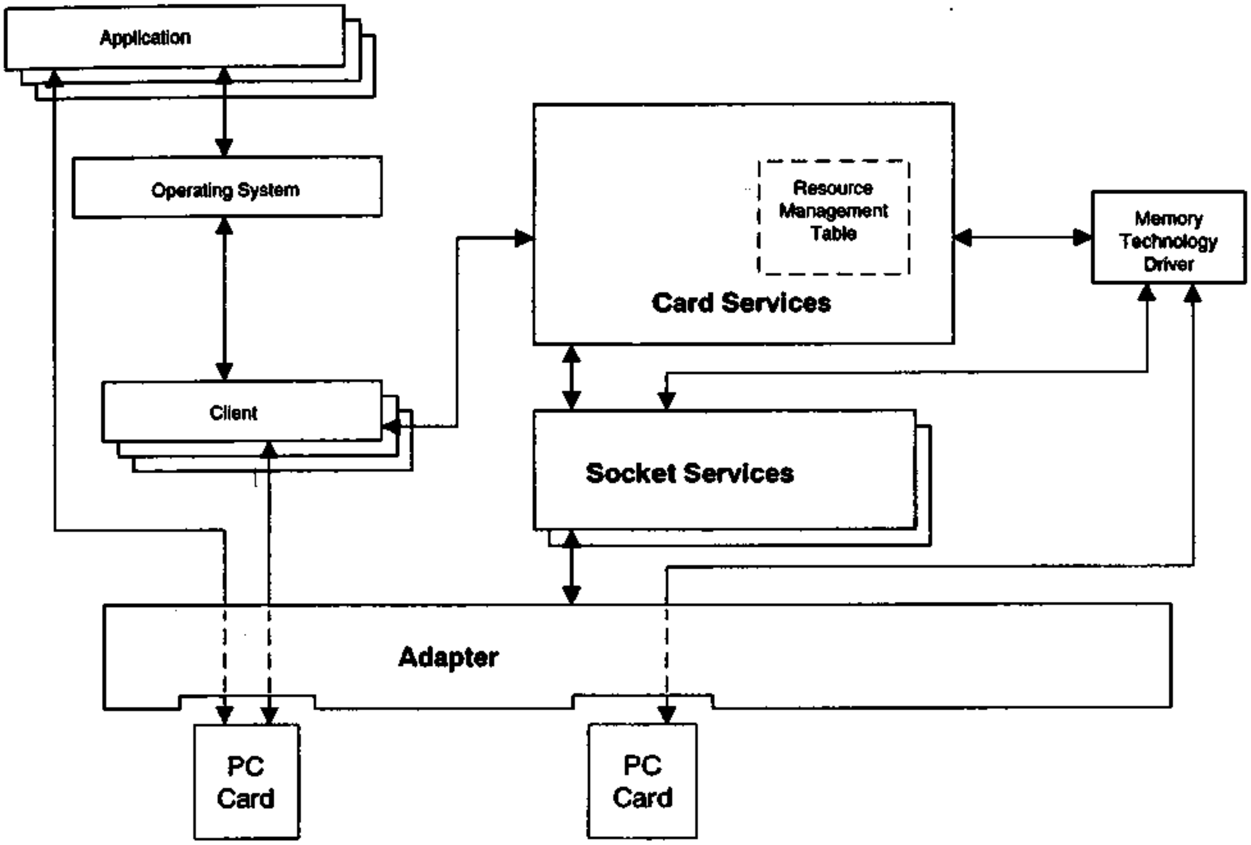
FUNCTIONAL DESCRIPTION

FUNCTIONAL DESCRIPTION

3.1 Architecture

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

Figure 3-1. Software Architecture Diagram



3.1.1 Hardware Layer (PC Cards, Sockets, and Adapters)

Cards compliant with the PCMCIA PC Card Standard are referred to as PC Cards. Release 1.0 of the standard specified data storage or memory cards. Release 2.01 of the standard expanded the definition of PC Cards to include peripheral expansion or I/O cards. Both types (memory and I/O) of these 68-pin PC Cards have the same physical characteristics and compatible electrical characteristics.

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

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

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

3.1.2 Socket Services

Immediately above the hardware layer the Socket Services software provides a standardized interface to manipulate PC Cards, sockets, and adapters. This interface is defined in the *PCMCIA Socket Services Interface Specification*.

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

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

3.1.3 Card Services

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

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

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

3.1.4 Memory Technology Drivers

The PCMCIA standard supports a wide range of memory devices on PC Cards. While all PC Cards may be read as if they contained static-RAM devices, special programming algorithms may be required to write or erase PC Cards. Card Services hides the details of what is required to write or erase PC Cards from client device drivers through byte-oriented write and copy functions and a block-oriented erase function.

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

Card Services provides default MTDs for recognized regions. If Card Services recognizes a region as being composed of Static RAM devices, it installs a default MTD that supports read and write requests. Reads and writes are performed as simple memory accesses without any algorithmic operation. Erase requests fail and return MEDIA_NOT_ERASABLE. If Card Services recognizes a memory region but not the type of devices in the region, it installs a default MTD that supports read and write requests and fails erase requests. The reads and writes are performed as simple memory accesses without any algorithmic operation. Card Services may include MTD support for other device types that require specific programming algorithms.

The interface between Card Services and MTDs is documented in Section 3.6.

3.1.5 Client Device Drivers

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

3.2 Programming Interface

3.2.1 Calling Conventions

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

3.2.1.1 Basic Operation

Card Services is invoked in a processor and Operating System dependent manner. See Appendix D for specific binding information.

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

- Function,
- Handle,
- Pointer,
- Status,
- Argument Length, and
- Argument Pointer.

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

```
status = CardServices(Function, Handle, Pointer, ArgLength, ArgPointer)
```

All requests pass the function code of the request in the Function argument. Individual functions and their function field values are described in later sections. Many requests require a Card Services handle to identify some resource. These requests pass the handle in the Handle argument. Some requests require an additional pointer value which is passed in the Pointer argument.

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

The ArgLength argument may be used by a Card Services implementation to validate that the argument packet is appropriate for the indicated function. In future releases, a request might define an extended length argument packet and Card Services could use this field to determine if the extended length argument packet was being used by the client requesting the function.

Appendix D defines the specific processor binding for the generic Card Services arguments.

3.2.1.2 Argument Packet

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

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

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

3.2.1.3 Logical Sockets

All Card Services functions, except **MapPhySocket**, use logical sockets to identify the socket a function is intended to address. The first physical socket on the first physical adapter is logical socket *zero (0)* ~~one (1)~~. The maximum logical socket is the total number of sockets present *minus one*.

3.2.1.4 Reserved Fields

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

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

3.2.1.5 Multi-Byte Fields

All multi-byte fields are stored in binding specific format. Multi-byte data returned in bulk from a PC Card is kept in little-endian format with the least significant byte appearing first in memory. For example, the **GetTupleData** and **Read/Write/CopyMemory** requests transfer the data without any byte swapping processing.

See the Appendix D for more information.

3.2.2 Presence Detection

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

3.2.3 Initialization of Card Services

Card Services is designed to be implemented as an Operating System Dependent Device Driver or OS extension. If a processor supports different modes of operation, Card Services can assume that it is used in only a single mode. For example, Intel 80386 compatible processors can run in Real Mode or Protect Mode. Card Services can assume that it is only used in one of these modes at any time.

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

Initialization is implementation specific.

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

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

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

3.2.4 Return Codes

Card Services indicates success or failure of a request with the generic Status argument. If the Status argument is set to a nonzero value on return from a Card Services request, the request failed and the value in the Status argument describes why the request failed. If the Status argument is reset to zero on return from a Card Services request, the request succeeded. See Appendix C or an individual function description for specific return code values. See Appendix D for the processor specific bindings of the Status argument.

3.3 Function Groups

Card Services functions may be divided into five functional groups:

- Client Services,
- Resource Management,
- Client Utilities,
- Bulk Memory Services, and
- Advanced Client Services.

Client Services.............................. provides for Client initialization and the callback registration of Clients.

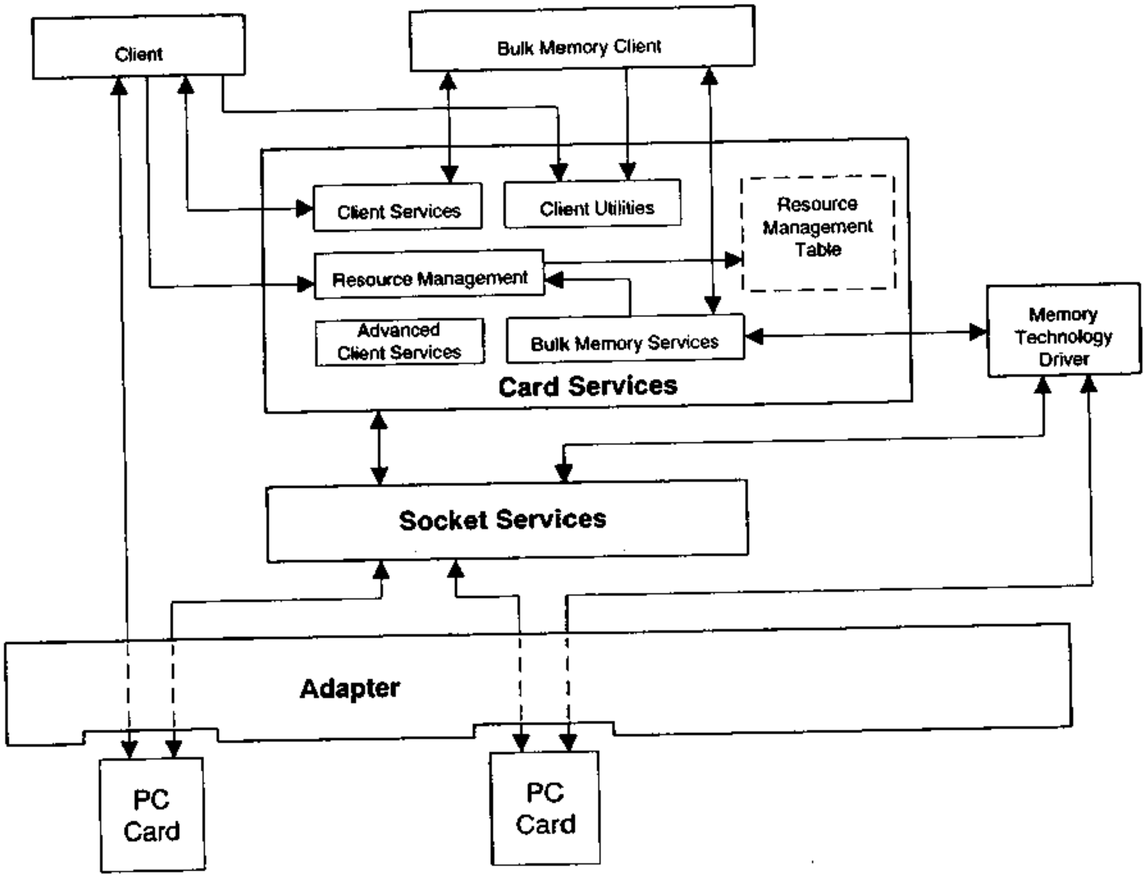
Resource Management............... provides basic access to available system resources, combining knowledge of the current status of system resources with the underlying Socket Services adapter control functions.

Client Utilities.............................. perform common tasks required by clients so that functions such as basic CIS tuple processing do not need to be duplicated in each of the client device drivers.

Bulk Memory Services............... provides read, write, copy and erase memory functions for use by file systems or other generic memory clients that want to be isolated from memory technology hardware details.

Advanced Client Services .... provide specific functions for clients with special needs.

Figure 3-2. Card Services Diagram



3.3.1 Client Services

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

- those that support client registration with Card Services to allow event notifications, and
- those that provide basic inquiry of PC Cards.

3.3.1.1 Client Registration

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

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

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

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

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

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

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

3.3.1.2 Basic Card Support

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

3.3.2 Resource Management

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

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

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

The **RequestSocketMask** function indicates that a client wishes to use the PC Card in a socket. This function allows the client to specify the events it is interested in monitoring. The **ReleaseSocketMask** function indicates that a client is no longer using a PC Card in a socket and is no longer interested in socket event notifications. This also allows Card Services to remove power from the socket when the last client releases the socket and no memory regions are open.

3.3.3 Bulk Memory Services

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

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

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

Performing an erase operation differs from the read, write and copy functions. A client that needs to erase memory must register an erase queue with the **RegisterEraseQueue** request. The client then fills an erase queue entry identifying the socket and region of memory to erase. Next, the **CheckEraseQueue** request is used to notify Card Services that one or more erase requests have been made in the erase queue. The actual erase operation is performed asynchronously. When the erase operation completes, the client is notified through the callback entry point provided in the erase queue header. In comparison, the read, write, and copy functions return only after the requested action has been completed.

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

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

3.3.4 Client Utilities

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

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

3.3.5 Advanced Client Services

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

WARNING:

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

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

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

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

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

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

AddSocketServices allows additional driver versions of Socket Services to be added to an already initialized Card Services. **ReplaceSocketServices** allows a different and potentially newer version of Socket Services to replace an existing version.

3.4 Callback Interfaces

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

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

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

- Function,
- Socket,
- Info,
- MTDRequest,
- Buffer,
- Misc,
- Status, and
- ClientData.

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

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

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

The Socket argument identifies the socket affected by the event. The **Info** argument contains other information specific to the event being reported. The **Status** argument is used by callback handlers to return information to Card Services. The **Buffer** argument is used to pass a buffer the client will fill for a **GetClientInfo** request. The **Misc** argument is used for miscellaneous information.

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

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

3.4.1 Insertion

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

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

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

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

3.4.2 Registration Completion

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

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

3.4.3 Status Change

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

BATTERY_LOW
CARD_LOCK
CARD_READY
BATTERY_DEAD
CARD_UNLOCK
CARD_REMOVAL

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

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

3.4.4 Ejection/Insertion Requests

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

EJECTION_REQUEST
INSERTION_REQUEST
EJECTION_COMPLETE
INSERTION_COMPLETE

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

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

3.4.5 Exclusive

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

EXCLUSIVE_REQUEST
EXCLUSIVE_COMPLETE

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

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

3.4.6 Reset

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

RESET_REQUEST
CARD_RESET
RESET_PHYSICAL
RESET_COMPLETE

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

3.4.7 Client Information

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

CLIENT_INFO

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

For the CLIENT_INFO event, the buffer contains the following fields:

Offset	Field	Size	Type	Value	Detail/Description
0	MaxLength	2	I	N	Maximum length of buffer
2	InfoLen	2	O	N	Length of Info returned by Client
4	Attributes	2	I/O	N	Bit-mapped
6	Revision	2	O	BCD	BCD Value of Vendor CS Revision
8	CSLevel	2	O	BCD	BCD Value of CS Release
10	RevDate	2	O	N	Revision Date in DOS Format
12	NameOff	2	O	N	Offset in packet to Client Name string
14	NameLen	2	O	N	Length of Client Name ASCIIZ string
16	VStringOff	2	O	N	Offset in packet to Vendor string
18	VStringLen	2	O	N	Length of Vendor ASCIIZ string
N	NameString	N	O	S	Client Name ASCIIZ string
N	VendorString	N	O	S	Vendor String ASCIIZ string

These fields are defined in the Function Reference for GetClientInfo.

3.4.8 Erase Completion

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

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

3.4.9 MTD Request

The MTD_REQUEST event passes Function, Buffer, MTDRequest, Socket and ClientData arguments to the client callback handler. See Section 3.6 for a description of the MTDRequest.

3.4.10 Timer

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

New Socket Services

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

The Info argument is bitmapped as follows:

Bits 0 -- 7	Number of sockets affected
Bit 8	New Sockets (set to one = true)
Bits 9 -- 15	RESERVED (reset to zero)

New Sockets bit is reset to zero if the previously installed Socket Services handler was replaced and is set to one if a new Socket Services handler is now providing support for additional sockets.

3.5 Events

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

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

3.5.1 BATTERY_DEAD

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

The BATTERY_DEAD event indicates the battery on a PC Card is no longer serviceable and data may be lost.

Cause The BATTERY_DEAD event occurs when the BVD1 signal on a PC Card is negated (asserted low). This signal may be available at the socket interface or in the pin replacement register. The negation of this signal results in a status change interrupt.

Pre-Client Card Services notes a transition to a BATTERY_DEAD state. When the Card Services interface is available, Card Services notifies clients who have indicated their interest in BATTERY_DEAD events.

Client A client processing BATTERY_DEAD notifications might warn the end-user that the PC Card is no longer capable of safely storing data if the PC Card is removed. How the client interacts with the end-user or what data loss preventive measures are taken is implementation specific.

Post-Client Card Services does not perform any additional processing after notifying clients using the socket of the BATTERY_DEAD event.

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

See also BATTERY_LOW.

3.5.2 BATTERY_LOW

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

The BATTERY_LOW event indicates the battery on a PC Card is weak and should be replaced. Data integrity on the PC Card is still assured.

Cause The BATTERY_LOW event occurs when the BVD2 signal on a PC Card is negated (asserted low). This signal may be available at the socket interface or in the pin replacement register. The negation of this signal results in a status change interrupt.

Pre-Client Card Services notes a transition to a BATTERY_LOW state. When the Card Services interface is available, Card Services notifies clients who have indicated their interest in BATTERY_LOW events.

Client A client processing BATTERY_LOW notifications might warn the end-user that the PC Card battery needs replacing, but data integrity is still assured. How the client interacts with the end-user or what data loss preventive measures are taken is implementation specific.

Post-Client Card Services does not perform any additional processing after notifying clients using the socket of the BATTERY_LOW event.

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

See also BATTERY_DEAD.

3.5.3 CARD_INSERTION

`Callback(CARD_INSERTION, Socket, 0, null, null, 0, ClientData)`

The CARD_INSERTION event indicates a PC Card has been inserted in a socket or a client has just registered for insertion events and Card Services is creating artificial insertion events for PC Cards already in sockets.

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

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

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

Pre-Client If power was not previously applied to the socket, Card Services sets Vcc to 5 volts. If the PC Card is not already in use, Card Services initiates a hardware reset of the PC Card. Sometime later, Card Services completes the reset.

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

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

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

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

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

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

Post-Client If no clients indicate they wish to use the socket with a **RequestSocketMask, OpenMemory**, or **RequestConfiguration** request after they have been notified of the CARD_INSERTION event, Card Services removes power from the socket.

See also CARD_REMOVAL.

3.5.4 CARD_LOCK

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

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

Cause Some sockets have hardware which can lock a PC Card into a socket to prevent inadvertent removal during operation. In addition, some sockets can report a change in the status of the locking hardware to warn that the card may be removed before it is actually removed. If a socket supports this capability, Card Services generates a CARD_LOCK event when the mechanical latch is locked.

Pre-Client Card Services does not perform any pre-client processing.

Client A client might respond to a CARD_LOCK event notification by setting internal state that it is safe to perform direct code execution from a PC Card. With the CARD_LOCK event, a client can be assured that directly executing code cannot be interrupted by the removal of a PC Card. Any client processing is implementation specific.

Post-Client Card Services does not perform any post-client processing.

See also CARD_UNLOCK.

3.5.5 CARD_READY

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

The CARD_READY event indicates a PC Card's +RDY/-BSY line has transitioned from the busy to ready state.

Cause A PC Card's +RDY/-BSY line changes from low to high.

Pre-Client Card Services does not perform any pre-client processing.

Client A client or MTD might respond to a CARD_READY event by completing an operation that is partially automated by a PC Card's on-board logic. It is expected that most clients will ignore CARD_READY events, performing polling to determine when PC Card's are in the ready state. Processing of this event is implementation specific.

Post-Client Card Services does not perform any post-client processing.

Note:

Most PC Card's drive the +RDY/-BSY line low (to the busy state) whenever data is output to the card. As soon as the PC Card is ready to receive additional data, the +RDY/-BSY line is returned to the high (ready) state. For that reason, CARD_READY events may be extremely frequent. Clients may completely ignore such events and improve overall system response by resetting the READY bit in their global and socket eventmasks by using SetEventMask as necessary.

3.5.6 CARD_REMOVAL

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

The CARD_REMOVAL event indicates a PC Card has been removed from a socket.

Cause The -CD1 and -CD2 pins in a socket transition from low to high or a client requests a PC Card be reset. The **RequestExclusive** function generates CARD_REMOVAL events to clients that were registered for the socket. The **ReleaseExclusive** function also generates a CARD_REMOVAL event to the client exclusively using a PC Card after a **RequestExclusive.**

Pre-Client Card Services does not perform any pre-client processing.

Client A client must perform corresponding **Release_** resource requests for all successfully performed **Request_** resource functions. **ReleaseConfiguration** must be requested first, followed by any additional **Request_** functions. For example, if a client has routed the PC Card's IREQ line with **RequestIRQ** then a **ReleaseIRQ** request should be made. This allows Card Services to update its internal database of system resource allocations and adjust socket hardware appropriately.

Post-Client Card Services removes power from the socket.

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

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

See also CARD_INSERTION and CARD_LOCK.

3.5.7 CARD_RESET

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

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

Cause A client requested a **ResetCard** and the reset has been completed.

Pre-Client Card Services has successfully performed a RESET_REQUEST notification and has physically reset the PC Card.

Client This is an opportunity for a client to re-establish any hardware state that existed before the PC Card was reset. The Info argument contains SUCCESS if the reset has been successfully completed. If the reset was rejected, the Info argument contains IN_USE. If the reset was not successful, Info contains a return code indicating the reason for the failure. Handling of the event is implementation specific.

Post-Client Card Services sends a RESET_COMPLETE notification directly to the client which requested the **ResetCard** function.

See also RESET_COMPLETE, RESET_PHYSICAL and RESET_REQUEST.

3.5.8 CARD_UNLOCK

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

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

Cause Some sockets have hardware which can lock a PC Card into a socket to prevent inadvertent removal during operation. In addition, some sockets can report a change in the status of the locking hardware to warn that the card can be removed before it is actually removed. If a socket supports this capability, Card Services generates a CARD_UNLOCK event when the mechanical latch is unlocked.

Pre-Client Card Services does not perform any pre-client processing.

Client A client might respond to a CARD_UNLOCK event notification by setting internal state that it is not safe to perform direct code execution from a PC Card. Processing of the event is implementation specific.

Post-Client Card Services does not perform any post-client processing.

See also CARD_LOCK.

3.5.9 CLIENT_INFO

```
status = Callback(CLIENT_INFO, 0, 0, null, Buffer, 0, ClientData)
```

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

Cause A requestor used **GetClientInfo** to ask Card Services to return information about a client.

Pre-Client Card Services calls the client for which information has been requested.

Client The client should copy its client information data into the buffer provided by Card Services.

Post-Client Card Services returns the client information data to the requestor.

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

See also CLIENT_INFOSIZE.

3.5.10 EJECTION_COMPLETE

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

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

Cause If a socket has hardware which can eject a PC Card from a socket, this event is generated when the ejection has been completed.

Pre-Client Card Services does not perform any pre-client processing.

Client A client might maintain an on-screen icon of the state of the socket. When this event is received, the icon could indicate ejection was complete and the PC Card could be removed from the socket. Processing of this even is implementation specific.

Post-Client Card Services turns off the ejection motor if this is not performed automatically.

See also EJECTION_REQUEST.

3.5.11 EJECTION_REQUEST

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

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

Cause If a socket has hardware which can eject a PC Card from a socket, this event is generated when an end-user requests ejection be performed.

Pre-Client Card Services does not perform any pre-client processing.

Client A client may ignore the event by returning with the Status argument set to SUCCESS. Card Services will then attempt to eject the PC Card from the socket. A client may also choose to prevent the ejection. In this case, the client should return from its callback handler with the Status argument not set to SUCCESS.

Post-Client Card Services either starts the ejection motor after all clients are notified or ignores the ejection request depending on the state of the Status argument on return from the client's callback handler. After the ejection motor has completed ejecting the PC Card, the EJECTION_COMPLETE event is generated.

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

See also EJECTION_COMPLETE, INSERTION_COMPLETE and INSERTION_REQUEST.

3.5.12 ERASE_COMPLETE

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

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

Cause The processing of a client's queued erase request has been completed by Card Services.

Pre-Client Card Services performs the processing required by the client's queued erase request.

Client The EraseQueueHandle is passed in the Misc argument. The EraseQueueEntryNum of the erase that was completed is passed in the Info argument. A client will check the EntryState for the affected erase queue entries to verify that the erase succeeded. The client then may immediately request that Card Services perform writes to record initial data structures in the newly erased block.

Post-Client Card Services does not perform any post-client processing.

3.5.13 EXCLUSIVE_COMPLETE

```
Callback(EXCLUSIVE_COMPLETE,   Socket,   ExclusiveStatus,   null,   null0,
         ClientData)
```

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

Cause A client uses **RequestExclusive** to gain exclusive access to a PC Card that may already be in use by other clients.

Pre-Client Card Services has completed its processing. The Info argument indicates the client now has exclusive use if set to SUCCESS. If Info is not set to SUCCESS, the **RequestExclusive** failed and the client does not have exclusive access to the PC Card and the Info value (return code) indicates the reason for failure.

Client If the request was successfully handled, the client can now use the PC Card exclusively.

Post-Client Card Services does not perform any post-client processing.

See also EXCLUSIVE_REQUEST.

3.5.14 EXCLUSIVE_REQUEST

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

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

Cause A client uses RequestExclusive to gain exclusive access to a PC Card that may already be in use by other clients. Card Services sends EXCLUSIVE_REQUEST events to clients registered for the affected PC Card. The clients use the event return code to indicate whether or not they are willing to relinquish use of the PC Card.

Pre-Client Card Services has already returned from the **RequestExclusive** function. This notification is being made from a background execution thread of Card Services. The Card Services interface is available.

Client If the client is willing to relinquish its use of the PC Card, it should return the status argument set to SUCCESS. If the client is not willing to relinquish its use of the PC Card, the status argument should not be set to SUCCESS.

Post-Client If any client rejects the event, Card Services terminates notification processing and notifies the requesting client that the exclusive request failed. Once all clients have accepted the EXCLUSIVE_REQUEST event, Card Services sends CARD_REMOVAL events to all clients registered and then sends a CARD_INSERTION event to the requesting client. Finally, Card Services sends the EXCLUSIVE_COMPLETE event to the requesting client.

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

See also EXCLUSIVE_COMPLETE.

3.5.15 INSERTION_COMPLETE

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

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

Cause If a socket has hardware which can insert a PC Card into a socket, this event is generated when the insertion has been completed.

Pre-Client Card Services does not perform any pre-client processing.

Client A client might maintain an on-screen icon of the state of the socket. When this event is received, the icon could indicate insertion was complete and the PC Card was in the socket. This functionality is implementation dependent.

Post-Client Card Services turns off the insertion motor, if this is not performed automatically.

See also INSERTION_REQUEST.

3.5.16 INSERTION_REQUEST

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

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

Cause If a socket has hardware which can insert a PC Card into a socket, this event is generated when an end-user requests insertion be performed.

Pre-Client Card Services does not perform any pre-client processing.

Client A client may ignore the event by returning with the Status argument set to SUCCESS. Card Services will then attempt to insert the PC Card into the socket. A client may also choose to prevent the insertion. In this case, the client should return from its callback handler with the Status argument not set to SUCCESS.

Post-Client Card Services starts the insertion motor after all clients are notified or ignores the request depending on the state of the Status argument on return from the client's callback handler. After the insertion motor has completed inserting the PC Card, the INSERTION_COMPLETE event is generated.

WARNING:

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

See also EJECTION_COMPLETE, EJECTION_REQUEST and INSERTION_COMPLETE.

3.5.17 PM_RESUME

This revision of Card Services does not specify how Power Management events are handled.

3.5.18 PM_SUSPEND

This revision of Card Services does not specify how Power Management events are handled.

3.5.19 REGISTRATION_COMPLETE

```
Callback(REGISTRATION_COMPLETE, 0, 0, null, null, 0, ClientData)
```

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

Cause The processing of a client's **RegisterClient** request has been completed by Card Services.

Pre-Client Card Services has completed notifying the client of any PC Cards that were already installed when the **RegisterClient** function was requested.

Client Clients have probably been waiting for this event to signal they may continue their foreground processes that generated the original request. In this case, clients will most likely just set a semaphore indicating the request is complete. Then, when their foreground process again receives control, it will confirm the semaphore has been set and continue processing. This functionality is implementation specific.

Post-Client Card Services does not perform any post-client processing.

3.5.20 RESET_COMPLETE

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

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

Cause The processing of a client's **ResetCard** request has been completed by Card Services.

Pre-Client Card Services has completed the reset processing for the specified card.

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

Post-Client Card Services does not perform any post-client processing.

3.5.21 RESET_PHYSICAL

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

The RESET_PHYSICAL event indicates a hardware reset is about to occur on the PC Card in the specified socket.

Cause A client requested a **ResetCard** and no client rejected the previous RESET_REQUEST event.

Pre-Client Card Services has successfully performed a RESET_REQUEST notification.

Client This is an opportunity for a client to save any hardware state that may be lost when the PC Card is physically reset. Client processing is implementation specific.

Post-Client Card Services sends a CARD_RESET to all clients and a RESET_COMPLETE notification directly to the client which requested the **ResetCard** function.

See also CARD_RESET, RESET_COMPLETE and RESET_REQUEST.

3.5.22 RESET_REQUEST

```
status = Callback(RESET_REQUEST, Socket, 0, null, null, 0, ClientData)
```

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

Cause A client has requested a **ResetCard** function.

Pre-Client Card Services has already returned to the client requesting the **ResetCard** function. This notification is being made from a background execution thread of Card Services. The Card Services interface is available.

Client This is an opportunity for a client to prevent the reset request from occurring. The Status argument indicates whether the client will allow the request to complete. If the Status argument is set to SUCCESS on return from client notification, Card Services continues to notify other clients. If the Status argument is not set to SUCCESS on return from client notification, Card Services sends a RESET_COMPLETE event to the requesting client with the Info argument indicating the request was rejected.

Post-Client Card Services sends a RESET_PHYSICAL notification and hardware reset is performed on the PC Card. Card Services then sends a CARD_RESET. Finally, Card Services sends a RESET_COMPLETE notification directly to the client which requested the **ResetCard** function.

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

See also CARD_RESET, RESET_COMPLETE and RESET_PHYSICAL.

3.5.23 TIMER_EXPIRED

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

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

Cause The wait count has expired for a **RegisterTimer** request.

Pre-Client Card Services has received a timer tick interrupt, noted the Card Services interface is available and the wait count is or will be decremented to zero (0) by this tick.

Client The client may perform any processing it may have delayed.

Post-Client No Post-Client processing is performed.

3.5.24 SS_UPDATED

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

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

Cause	**AddSocketServices** or **ReplaceSocketServices** has been called.
Pre-Client	Card Services uses the new Socket Services handler to determine what hardware resources are now available for the affected sockets.
Client	The client may react to the new or changed socket support.
Post-Client	No Post-Client processing is performed.

3.5.25 WRITE_PROTECT

`Callback(WRITE_PROTECT, Socket, WPState, NULL, NULL, NULL, ClientData)`

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

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

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

3.6 Memory Technology Drivers

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

3.6.1 Registration

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

When an MTD is notified of CARD_INSERTION events, it can use the **GetFirst/NextRegion** functions to determine if it wishes to handle read, write, copy and erase requests for memory on the PC Card. If an MTD elects to handle a region, it performs a **RegisterMTD** request to inform Card Services to use the MTD for all access to the region. If an MTD elects not to handle a region, the region is then handled by a previously installed MTD. By default, Card Services installs an

MTD that supports read and write access to SRAM memory regions and non SRAM regions. In general, a client cannot depend on the state of memory after an erase request for a given MTD—the value read from a memory area that was just erased is undefined.

3.6.2 Card Services/MTD Interface

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

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

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

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

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

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

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

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

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

TransferLength contains the length in bytes of the request.

Function is a bit-mapped field defined as:

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

Command identifies the MTD function request.

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

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

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

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

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

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

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

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

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

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

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

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

3.6.3 MTD Helper Interface

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

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

3.6.4 Erase Queuing

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

3.6.5 Blocking

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

3.6.6 Card Services Request Retries

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

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

- POWER,
- MTD,
- TIMER, and
- READY.

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

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

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

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

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

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

3.6.7 Media Access Table

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

The entry points are ordered as follows:

Function	Description
MATData	Pointer to the MAT data area
CardSetAddress	Establishes access to a PC Card memory area
CardSetAutoInc	Enables autoincrementing addresses
CardReadByte	Reads a byte from the memory area
CardReadWord	Reads a word from the memory area
CardReadWords	Reads words from the memory area. For use with AIMS PC Cards.
CardReadByteAI	Reads a byte from the memory area and automatically increments the memory address to the next byte.

CardReadWordAI........................Reads a word from the memory area and automatically increments the memory address to the next word.

CardReadWordsAI......................Reads a block of memory incrementing the memory address.

CardWriteByte..............................Writes a byte to the memory area

CardWriteWord ....Writes a word to the memory area

CardWriteWords..........................Writes words to the memory area. For use with AIMS PC Cards.

CardWriteByteAI ....Writes a byte to the memory area and automatically increments the memory address to the next byte.

CardWriteWordAI.......................Writes a word to the memory area and automatically increments the memory address to the next word.

CardWriteWordsAI ....Writes a block of memory incrementing the memory address.

CardCompareByte ....Compares a byte with a byte in PC Card memory.

CardCompareByteAI..................Compares a byte incrementing the memory address.

CardCompareWords ....Compares a block of memory against a block of PC Card memory.

CardCompareWordsAI...............Compares a block of memory incrementing the memory address.

MTDHelperEntry........................The entry point for MTD helper functions of Card Services.

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

3.6.8 Virtual Memory Partitions/Regions

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

MTDs can use special Card Services features to allow clients to use the **Open / Close / Read / Write / Copy / EraseMemory** requests to access such memory. **Get First/Next Region/Partition** will return information about such memory areas to requesting clients.

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

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

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

3.6.9 Tuple Usage

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

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

During pre-client processing of CARD_INSERTION events, Card Services identifies all of the regions present on a PC Card. This may require processing multiple Device Information and JEDEC Identifier tuples.

The tuple names listed above are from the *PCMCIA PC Card Standard*, Release 2.01. Specific tuple codes may be determined by reference to the appropriate documents.

SECTION - 4

ASSUMPTIONS AND CONSTRAINTS

ASSUMPTIONS AND CONSTRAINTS

4.1 Power Management

This revision of Card Services does not provide an interface for power management.

4.2 Auto Configuration of I/O Cards

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

4.3 Compression

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

4.4 EDC Generation

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

4.5 BIOS or Device Driver

Card Services can be ROM-able. Card Services is intended to be an Operating System dependent loadable device driver or OS extension. During initialization, Card Services allocates its own RAM. Card Services may be dependent on specific details of the hardware, e.g. whether the system bus is ISA, EISA, or Microchannel.

4.6 Interrupts Per Socket

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

4.7 Mixed Media Memory Cards

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

4.8 Multiple Partitioned Memory Cards

Card Services should be implemented to support more than one partition on a PC Card or within a region. The number of partitions supported per PC Card is implementation specific.

4.9 Use of Socket Services

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

4.10 Interface Assumptions

4.10.1 Range Checking of Arguments

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

4.10.2 Configuration

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

4.10.3 Abnormal Termination

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

4.11 Timeouts

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

4.12 CIS Access

It is possible that a client could set a mode with some PC Cards that makes CIS unavailable. If a PC Card is to be used by more than one client, the card must allow the CIS to be read in any possible configuration since there is no way to guarantee in what order clients will initialize the card.

SECTION - 5

FUNCTION REFERENCE

FUNCTION REFERENCE

The following sections describe the Card Services functions in detail. The functions are listed alphabetically.

Each function is listed with its calling arguments identified for ease of reference. The functional notation used for a Card Services call is:

```
CardServices(Function, Handle, Pointer, ArgLength, ArgPointer)
```

For example:

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

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

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

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

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

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

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

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

5.0 AccessConfigurationRegister (36H)

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

This function allows a client to read or write a PC Card Configuration Register.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Action	1	I	N	Read/Write operation
3	Offset	1	I	N	Offset to Configuration Register
4	Value	1	I/O	N	Value to read or to write

*The **Socket** field identifies the logical socket for the PC Card to access.*

*The **Action** field may be set to READ (00H) or WRITE (01H). All other values in the **Action** field are reserved for future use. If the **Action** field is set to WRITE, the Value field is written to the specified Configuration Register. Card Services does not read the Configuration Register after a write operation. For that reason, the **Value** field is only updated by a read request.*

*The **Offset** field specifies the byte offset for the desired configuration register from the PC Card configuration register base specified in **RequestConfiguration**.*

*The **Value** field contains the value read from the PC Card configuration register for a read operation. For a write operation, the **Value** field contains the value to write to the configuration register. As noted above, on return from a write request, the **Value** field is the value written to the PC Card and not any changed value that may have resulted from the write request (i.e. no read after write is performed).*

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

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

Return Codes:

BAD_ARG_LENGTH	Arglength is not equal to five (5)
BAD_ARGS	Specified arguments are invalid
BAD_SOCKET	Socket is invalid
UNSUPPORTED_FUNCTION	This function is not supported

5.1 AddSocketServices (32H)

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

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

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

The **Attributes** field defines details about the new Socket Services entry point. The definition is binding specific.

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

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

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

Return Codes:

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

5.2 AdjustResourceInfo (35H)

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

~~This function makes adjustments to t~~The Card Services internal database of system resources that *are available for allocation to clients is managed by this function. The function is used to adjust or inquire about the system resource availability.* ~~can be allocated for use by clients. This function will not be typically used by clients, but can be used to adjust the system resources that Card Services manages~~.

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

The **Action** field has the following defined values:

0	Remove*ManagedResource*
1	Add*ManagedResource*
2	*GetFirstManagedResource*
3	*GetNextManagedResource*
4 -- 255	RESERVED ~~(reset to zero)~~

~~Remove indicates the resource is being used by some other entity in the system and should not be used by Card Services. Add indicates the resource is available for use by Card Services.~~

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

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

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

~~The Resource field indicates the type of resource that is being adjusted~~ *The **Resource** field identifies the system resource type:*

0	~~System Mappable~~ Memory Range
1	~~System Mappable~~ I/O Range
2	~~System Steerable~~ IRQ
3 -- 255	RESERVED ~~(reset to zero)~~

The remainder of the argument packet is structured based on the ~~particular~~ *system* resource *type* ~~being adjusted~~.

For *Memory Range resource types* ~~a System Mappable Memory Range resource adjustment~~ the argument packet has the following fields.

Offset	Field	Size	Type	Value	Detail/Description
0	Action	1	I	N	Add/Remove/*Query* resource
1	Resource	1	I	*0* ~~N~~	*Memory Range resource* ~~The resource type to adjust~~
2	Attributes	2	I/O	N	Attributes of the memory range
4	Base	4	I/O	N	System ~~b~~Base ~~a~~Address
8	Size	4	I/O	N	Memory ~~w~~Window ~~s~~Size

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

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

Shared is set to one if the memory range is being used but *is sharable* ~~can be shared~~ by other clients.

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

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

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

The **Size** field is the ~~byte~~ size of the ~~system~~ memory range *in bytes* ~~being adjusted~~.

Return Codes for memory adjustments:

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

For ~~a System-Mappable~~ I/O Range resource *types* ~~adjustment~~ the argument packet has the following fields.

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

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

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

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

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

Shared is set if the I/O range is being used but *is shareable* ~~can be shared~~ by other clients.

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

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

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

Return Codes for I/O *Range resource types are* ~~**adjustments:**~~

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

For ~~an System Steerable~~ IRQ *Level* resource *types* ~~adjustment~~ the argument packet has the following fields.

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

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

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

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

IRQ Type is set to Time-Multiplexed Shared if the the IRQ is *in use* ~~being used~~ but *is shareable* ~~can be shared~~ with other clients *with only one client using the IRQ at a time*~~in a time multiplexed fashion~~. **IRQ Type** is set to Dynamic Shared if the IRQ is being used but *is actively shareable* ~~can be shared~~ with other clients *at all times*~~with appropriate hardware support~~.

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

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

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

Return Codes for IRQ ***Level resource types are*****~~adjustments:~~**

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

5.3 CheckEraseQueue (26H)

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

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

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

Return Codes:

BAD_HANDLE	Invalid erase queue handle

5.4 CloseMemory (00H)

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

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

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

Return Codes:

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

5.5 [illegible] CopyMemory (01H)

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

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

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

The **Source Offset** is the relative starting location on the PC Card where the data to be copied originates.

The **Dest Offset** is the relative starting location on the PC Card where the data is to be placed.

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

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

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

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

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

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

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

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

Return Codes:

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

5.6 DeregisterClient (02H)

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

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

The client must have returned all requested resources before this function is called. If any resources have not been released, IN_USE is returned.

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

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

WARNING:

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

See also RegisterClient.

Return Codes:

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

5.7 DeregisterEraseQueue (25H)

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

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

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

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

See also **RegisterEraseQueue.**

Return Codes:

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

5.8 GetCardServicesInfo (0BH)

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

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

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

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

~~The Signature field is returned by this function. It is set to the ASCII characters 'CS' by Card Services if Card Services is installed.~~ *The **Signature** fields are returned as two ASCII characters, with **Signature[0]** set to the ASCII character 'C' (43H) and **Signature[1]** set to the ASCII character 'S' (53H). This **Signature** fields should* ~~It must~~ be reset to zero (0) before this function is invoked *to prevent false sensing.*

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

The **Count** field returns the number of logical sockets managed by Card Services. This value may be zero (0), if no sockets are present. Logical sockets are numbered from one (1) to the value returned in the **Count** field. Determining which physical adapter and socket correspond to a logical socket number may be done with the **MapLogSocket** request.

The **VendorString** field is an ASCIIZ string describing the Card Services implementor. It is expected to be used by system utilities for display purposes. The offset of the string from the beginning of the argument packet is specified in **VStrOff.** The actual length of the string (including the terminating zero) is returned in the **VStrLen** field. The string may include copyright legends and may be formatted with carriage return and linefeed characters. If the **VStrLen** field is zero, the ASCIIZ string describing the implementor is not present. The **Revision** field is the vendor's internal revision number for this specific implementation of Card Services. It is stored as a BCD value with an implied decimal point (e.g. Revision 3.10 would be 310H.). The **CSLevel field** indicates the level of compliance with an Card Services Release number. It is stored as a BCD value with an implied decimal point (e.g. Release 1.00 would be 100H).

Return Codes:

BAD_ARG_LENGTH	ArgLength is less than four (4)
UNSUPPORTED_FUNCTION	Card Services not installed

5.9 GetClientInfo (03H)

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

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

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

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

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

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

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

Offset	Field	Size	Type	Value	Detail/Description
0	MaxLen	2	I	N	Length of this packet
2	InfoLen	2	O	N	Length of Info returned by Client
4	Attributes	2	I/O	N	Bit-mapped (defined below)
6	Client Info	N	O	N	Client Information
6	Revision	2	O	BCD	BCD Value of Vendor client Revision
8	CSLevel	2	O	BCD	BCD Value of CS Release
10	RevDate	2	O	N	Revision Date
12	NameOff	2	O	N	Offset to ClientName String
14	NameLen	2	O	N	Length of Client Name ASCIIZ string
16	VStringOff	2	O	N	Offset in packet to Vendor string buffer
18	VStringLen	2	O	N	Length of Vendor ASCIIZ string
N	NameString	N	O	N	Client Name ASCIIZ string
N	VendorString	N	O	N	Vendor String ASCIIZ string

The **Revision** field is the vendor's internal revision number for this specific implementation of the client. It is stored as a BCD value with an implied decimal point (e.g. Revision 2.03 would be 203H). The **CSLevel** field indicates the compliance level with a Card Services release number. It is stored as a BCD value with an implied decimal point. (e.g. Release 1.00 would be 100H). The **RevDate** field describes the revision date of the client implementation. It is stored packed in the same manner as dates in an MS-DOS directory entry. This format is:

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

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

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

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

If the Info SubFunction value is 80H - 0FFH the Client Info field is structured according to that client specific subfunction. The subfunction codes 1 - 7FH are reserved for future Card Services functions. Card Services only ensures that the data returned by the replying client is returned to the requesting client and does not interpret or modify any such data.

Return Codes:

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

5.10 GetConfigurationInfo (04H)

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

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

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	O	N	Bit-mapped
4	Vcc	1	O	N	Vcc Setting
5	Vpp1	1	O	N	Vpp1 Setting
6	Vpp2	1	O	N	Vpp2 Setting
7	IntType	1	O	N	Memory or Memory+I/O Interface
8	ConfigBase	4	O	N	Card Base address of config registers
12	Status	1	O	N	Card Status register setting, if present
13	Pin	1	O	N	Card Pin register setting, if present
14	Copy	1	O	N	Card Socket/Copy register setting, if present
15	Option	1	O	N	Card Option register setting, if present
16	Present	1	O	N	Card Configuration registers present
17	FirstDevType	1	O	N	From Device ID Tuple
18	FuncCode	1	O	N	From Function ID Tuple
19	SysInitMask	1	O	N	From Function ID Tuple
20	ManufCode	2	O	N	From Manufacturer ID Tuple
22	ManufInfo	2	O	N	From Manufacturer ID Tuple
24	CardValues	1	O	N	Valid Card Register Values
25	AssignedIRQ	1	O	N	IRQ assigned to PC Card
26	IRQAttributes	2	O	N	Attributes for assigned IRQ
28	Base Port1	2	O	N	Base port address for range
30	Num Ports1	1	O	N	Number of contiguous ports
31	Attributes1	1	O	N	Bit-mapped
32	Base Port2	2	O	N	Base port address for range
34	Num Ports2	1	O	N	Number of contiguous ports
35	Attributes2	1	O	N	Bit-mapped
36	IOAddrLines	1	O	N	Number of IO address lines decoded

The fields from **Socket** to **IntType** are the same fields as for the **RequestConfiguration** function. The **Attributes** field additionally has *the following bits* ~~bit 0~~ defined~~-as~~:

Bit 0	Exclusively Used (set = true)
Bit 6	*Valid Client (set = true)*

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

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

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

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

The **FirstDevType, FuncCode, SysInitMask, ManufCode,** and **ManufInfo** fields are the values from the corresponding PCMCIA Tuples found on the PC Card. A value of 0FFH in any of the above tuple fields indicates that the corresponding tuple is not in the CIS on the PC Card.

The **CardValues** field indicates which of the Card Configuration register values were written to the PC Card. If the PC Card was configured during POST by the BIOS, Card Services may not know what values were written to the PC Card registers. The field is bit-mapped as follows:

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

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

The **BasePorts1** to **IOAddrLines** fields are the same as defined in **RequestIO.** *If the socket is not configured to use I/O ports **(RequestIO()** has not been successfully invoked), the **NumPorts1** and NumPorts2 fields are set to 00H.*

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to thirty-seven (37)
BAD_SOCKET	Socket is invalid
NO_CARD	No PC Card in socket

5.11 GetEventMask (2EH)

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

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

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

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

Bit 0	Event mask of this socket only (set = true)
Bits 1 -- 15	RESERVED (Reset to zero)

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

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

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

The **Socket** field identifies the logical socket when **Attributes** Bit 0 is set.

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

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to size (6)
BAD_HANDLE	ClientHandle is invalid
BAD_SOCKET	Socket is invalid (socket event mask, only)
NO_CARD	No PC Card in socket

5.12 GetFirstClient (0EH)

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

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

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

The **Socket** field identifies the logical socket if **Attributes** Bit 0 is set to one (1).

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

Bit 0	All clients (reset to zero) or clients for this socket only (set to one).
Bits 1 -- 15	RESERVED (Reset to zero)

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

WARNING

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

Return Codes:

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

5.3 GetFirstPartition (05H)

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

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

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

The **Socket** field describes the logical socket containing the desired card.

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

Bit 0	Memory type (set = attribute, reset = common)
Bit 1 - 7	RESERVED (Reset to zero)
Bit 8	Virtual Partition (set to one = true)
Bits 9 - 10	Write/Erase interactions: 0 - Write without Erase 1 - Write with Erase 2 - Reserved 3 - Write with Disableable Erase
Bit 11	Write with Verify
Bit 12	Erase Requests Supported
Bits 13 - 15	RESERVED (Reset to zero)

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

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

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

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

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

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

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero (noted as reserved in the *PCMCIA PC Card Standard - Release 2.0*), the lower bits are a binary code representing a speed from the following table:

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

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

The **CardOffset** field is set by Card Services. It is the offset on the card where this partition begins.

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

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

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

The lower fifteen bits of the **Part Type** field is a constant describing the type of partition. The following are currently defined:

0000H	No partition information tuple
0001H	DOS BPB/FAT partition
0002H	FFS I partition
0003H	FFS II partition
0004H	XIP partition
7FFFH	Unknown partition
	All other values are *RESERVED.*

The upper bit of the **Part Type** field indicates if the partition has EDC information. It will be set if the error detection code type in the TPLFMT_EDC field of the CISTPL_FORMAT tuple is non-zero. Clients may ignore the partition if they are not prepared to deal with EDCs. If a client can handle EDCs, it should use the Card Information Structure processing functions to recover detailed EDC information. The client must first locate the appropriate format tuple and then process the EDC information.

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

WARNING:

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

5.14 GetFirstRegion (06H)

```
CardServices(GetFirstRegion, null/MTDhandle, null, ArgLength, ArgPointer)
```

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

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

The fields in this argument packet are the same as in **GetFirstPartition**. The fields italicized in the argument packet described above are for internal use by Card Services, Card Services initialized them to the appropriate values. The client must preserve these values for subsequent **Get*Next*~~First~~Region** requests. The **Virtual Partition** attribute bit is interpreted as a virtual region attribute for regions.

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

Return Codes:

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

5.15 GetFirstTuple (07H)

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

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

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

The **Socket** field describes the logical socket containing the desired card.

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

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

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

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

The **TupleCode** and **TupleLink** fields are the values returned from the tuple found.

Return Codes:

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

5.16 GetNextClient (2AH)

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

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

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

The Socket field describes the logical socket if Attributes Bit 0 is set to one (1).

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

Bit 0	All clients (reset to zero) or clients for this socket only (set to one).
Bits 1 -- 15	RESERVED (Reset to zero)

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

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

Return Codes:

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

5.17 GetNextPartition (08H)

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

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

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

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

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

WARNING:

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

Return Codes:

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

5.18 GetNextRegion (09H)

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

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

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

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

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

Return Codes:

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

GetNextTuple (0AH)

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

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

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

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

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

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

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

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

The TupleCode and TupleLink fields are the values returned from the tuple found.

GetTupleData can be used to retrieve the actual tuple data.

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

ll, ArgLength, Ar

devices on the card in
ions, the Status argum
; access to this region is

etail/Description
ogical Socket
egion Attributes Field
t-mapped field
indow Speed Field
S Partition Flags Data
S Partition Link Data
S Partition CIS Data
rd Memory Region Offset
gion Size
se Block Size
rtition Multiple (Erase Block unit
tition JEDEC Memory ID Code

. Card Services initi
necessary, Card Servi
etween GetNextRegio
ion request. The Fla
te information for
ds are used by Card
sts. The Attributes fi

her return codes are t

l to thirty-two (32)
st/Next is corrupt

Return Codes:

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

5.20 GetStatus (0CH)

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

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

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

The **Socket** field identifies the logical socket which contains the PC Card.

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

Bit 0	Write Protected (set = true)
Bit 1	Card Locked
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Voltage Detect 1 (set = dead)
Bit 5	Battery Voltage Detect 2 (set = warning)
Bit 6	Ready/Busy (set = ready)
Bit 7	Card Detected (set = true)
Bits 8 -- 15	RESERVED (Reset to zero)

This information is obtained from the Socket Services **GetStatus** function. The **CardState field is** created from the Socket Services **Card State**. If an I/O card is installed in the specified socket, card state is returned from the pin replacement register. If certain state bits are not present in the pin replacement register (see PC Card Release 2.0 Section 5.2.8.3.2), a simulated state **bit** value is returned as defined below:

WP (Write Protected)	Not write protected
BVD1 (Battery Voltage Detect 1)	Power Good
BVD2 (Battery Voltage Detect 2)	Power Good
RDY/BSY (Ready/Busy)	Ready

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

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

This information is obtained form the Socket Services **Get Socket** function for memory cards. The **SocketState** field is created from the Socket Services **Socket Attributes**.

SUCCESS is returned if the Socket field is valid.

> **Note:**
> **NO_CARD will not be returned if there is no PC Card present in the socket.**

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to six (6)
BAD_SOCKET	Socket is invalid

5.21 GetTupleData (0DH)

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

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

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

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

The **Socket** field identifies the logical socket containing the PC Card.

The **Attributes, DesiredTuple, Flags, LinkOffset,** and **CISOffset** fields (italicized above) are for internal use by Card Services. They must be the same values returned by a **GetFirstTuple** or previous **GetNextTuple** request. Their exit values should be preserved by the client for subsequent **GetNextTuple** requests.

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

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

> ~~**Note:**~~
> ~~**NO_CARD will not be returned if there is no PC Card present in the socket.**~~

Return Codes:

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

5.22 MapLogSocket (12H)

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

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

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

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

The **Phy Adapter** field is returned by Card Services, if the **Log Socket** field is valid. It is the Socket Services adapter value to address the logical socket. *Physical adapters and physical sockets* ~~Adapters~~ are numbered starting at *zero* (*0*) ~~one (1) while sockets are numbered starting at one (1)~~. The **Phy Socket** field is returned by Card Services, if the **Log Socket** field is valid. It is the Socket Services socket value to address the logical socket.

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

Return Codes:

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

5.23 MapLogWindow (13H)

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

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

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

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

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

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to two (2)
BAD_HANDLE	WindowHandle is invalid

5.24 MapMemPage (14H)

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

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

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

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

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

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to five (5)
BAD_HANDLE	WindowHandle is invalid
BAD_OFFSET	Offset is invalid
BAD_PAGE	Page is invalid
NO_CARD	No card in socket

5.25 MapPhySocket (15H)

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

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

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

The **Log Socket** field is returned by Card Services. It contains the logical socket representing the **Phy Adapter** and **Phy Socket** provided. *Physical adapters and physical sockets* ~~Adapters~~ are numbered starting at *zero* (*0*)~~-one (1) while sockets are numbered starting at one (1)~~. The **Phy Adapter** field and **Phy Socket** fields, if valid, are converted to a **Log Socket** value.

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

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

Return Codes:

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

5.26 MapPhyWindow (16H)

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

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

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

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

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

Return Codes:

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

5.27 ModifyConfiguration (27H)

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

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

IO addresses mapped and IRQ routing can only be changed by first using **ReleaseConfiguration** and then using **Release/RequestIO/IRQ** followed by **RequestConfiguration.**

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Vcc	1	I	N	Vcc Setting
5	Vpp1	1	I	N	Vpp1 Setting
6	Vpp2	1	I	N	Vpp2 Setting

The **Socket** field must be the same value used in **RequestConfiguration.**

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

Bit 0	RESERVED (Reset to zero)
Bit 1	Enable IRQ Steering (set = true)
Bit 2	IRQ change valid (set = true)
Bit 3	Vcc change valid (set = true)
Bit 4	Vpp1 change valid (set = true)
Bit 5	Vpp2 change valid (set = true)
Bits 6 -- 15	RESERVED (Reset to zero)

Enable IRQ Steering is set to one to connect the PC Card IREQ to a previously selected system interrupt. **IRQ change valid** is set to one to request the IRQ steering enable to be changed. The **Vcc/Vpp1/Vpp2 change valid** bits are set to one to request a change to the corresponding voltage level for the PC Card.

The **Vcc, Vpp1** and **Vpp2** fields all represent voltages expressed in tenths of a volt. Since these fields are a byte wide, values from zero (0) to 25.5 volts may be set. To be valid, the exact voltage must be available through the system's Socket Services. To be compliant with the *PCMICA PC Card Standard - Release 2.0,* systems must always support 5.0 volts for both Vcc and Vpp.

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

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to seven (7)
BAD_ATTRIBUTE	IRQ steering cannot be disabled or enabled
BAD_HANDLE	ClientHandle does not match owning client
BAD_SOCKET	Socket is invalid
BAD_VCC	Requested Vcc is not available on socket
BAD_VPP	Requested Vpp is not available on socket
NO_CARD	No PC Card in socket

5.28 ModifyWindow (17H)

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

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

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

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

Bit 0	RESERVED (Reset to zero)
Bit 1	Memory type (set = attribute)
Bit 2	Enable (set = true, reset = disable)
Bit 3	AccessSpeed valid (set = true)
Bits 4 -- 15	RESERVED (Reset to zero)

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

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

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

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero (noted as reserved in the *PCMCIA PC Card Standard - Release 2.0)*, the lower bits are a binary code representing a speed from the following table:

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

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

Note:

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

Return Codes:

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

5.29 OpenMemory (18H)

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

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

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

The **Socket** field describes the logical socket containing the desired card.

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

Bit 0	Memory type (set = attribute)
Bit 1	Exclusive (set = true)
Bits 2 -- 15	RESERVED (Reset to zero)

Bit 0 specifies whether attribute or common memory will be accessed. If attribute memory is being accessed, the client must explicitly access the memory correctly. For PCMCIA defined attribute memory, this means that only even bytes can be reliably read and written. Bit 1 allows a client to gain exclusive access to the memory area beginning at the specified offset. Other clients that attempt to open the memory area beginning at the same offset will receive an IN_USE return code.

The **Offset** identifies the byte offset to the beginning of the portion of the card that the client will be accessing. Card Services uses this information to determine the correct MemoryHandle to return. The offset is also saved by Card Services to adjust relative offsets supplied by the **Read/Write/CopyMemory** and the erase functions to absolute offsets for MTDs.

The **MemoryHandle** field is returned by this request. It must be used for all subsequent read, write, copy and erase requests to the identified memory area. When all accesses have been performed, the client must perform a **CloseMemory** request with this handle. Each **OpenMemory** increments a use count maintained for each region and each **CloseMemory** decrements this counter.

Only one **OpenMemory** needs to be performed for all accesses to a specific area of a PC Card by a single client.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to eight (8)
BAD_HANDLE	Invalid ClientHandle
BAD_OFFSET	Offset is invalid
BAD_SOCKET	Socket Is Invalid
NO_CARD	No PC Card in socket
IN_USE	Memory area is in-use, exclusively

5.30 *ReadMemory (19H)* ~~5.31 RegisterClient (10H)~~

`CardServices(ReadMemory, MemoryHandle, buffer, ArgLength, ArgPointer)`

This function reads data from a PC Card via the specified MemoryHandle. An MTD is used to perform the actual read. The MemoryHandle returned by **OpenMemory** is passed in the Handle argument. The pointer to the system memory buffer that will receive the data read from the PC Card is passed in the Pointer argument.

Offset	Field	Size	Type	Value	Detail/Description
0	CardOffset	4	I	N	Card Source Offset
4	Count	4	I	N	Number of bytes to transfer

The **Card Offset** is a relative offset from the physical offset provided to the **OpenMemory** request used to obtain the **MemoryHandle**. It is the first location on the PC Card where the data should be read.

The **Count** field is the number of bytes to read from the PC Card.

If the **MemoryHandle** identifies common memory, all bytes requested are transferred. If the **MemoryHandle** identifies attribute memory, the client must recognize that memory locations with odd addresses may not have valid data. The client is expected to access the attribute memory in an appropriate way.

When used in a processor mode that supports segmentation (e.g. Intel 80286 protected mode operation), all bytes transferred are required to be contained within the segment referenced by the system memory buffer pointer. This limits the count requested to be less than or equal to the maximum segment size.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to eight (8)
BAD_HANDLE	Invalid MemoryHandle
BAD_OFFSET	Invalid source offset
READ_FAILURE	Unable to read media
BAD_SIZE	Size of area to read is invalid
NO_CARD	No PC Card in socket

5.31 *RegisterClient (10H)* ~~5.30 ReadMemory (19H)~~

```
CardServices(RegisterClient, null/ClientHandle, ClientEntry, ArgLength,
             ArgPointer)
```

This function registers a client with Card Services. A REGISTRATION_COMPLETE event is generated for the client callback handler when registration processing for the client has been completed by Card Services. The ClientHandle returned in the Handle argument must be passed to **DeregisterClient** when the client terminates. The Client callback handler entry point is passed in the Pointer argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Bit-mapped (defined below)
2	Event Mask	2	I	N	Events to notify Client
4	Client Data	8	I	N	Data for the Client (binding specific)
12	Version	2	I	N	Card Services Version this client expects

The Attributes field is bit-mapped. It identifies the type of client registering and what type of artificial CARD_INSERTION notifications Card Services should generate. The field is defined as follows:

Bit 0	Memory client device driver (set = true)
Bit 1	Memory Technology Driver (set = true)
Bit 2	IO client device driver (set = true)
Bit 3	CARD_INSERTION events for sharable PC Cards (set = true)
Bit 4	CARD_INSERTION events for cards being exclusively used (set = true)
Bits 5 -- 15	RESERVED (reset to zero)

Bits 0, 1 and 2 are mutually exclusive, only one bit may be set to one, but one of the bits must be set to one. If both bits 3 and 4 are reset to zero, the client will not receive artificial CARD_INSERTION notifications and also will not receive a REGISTRATION_COMPLETE notification.

I/O client device drivers are notified of card insertions before other clients. IO clients are notified in LIFO order (i.e. the last IO client to register is notified first) to ensure that the most recent, and presumed up to date, client is the one that sets the interface for the PC Card and socket. Memory Technology Drivers are notified of card insertions next in FIFO order (i.e. the first MTD to register is notified first). Finally, memory client device drivers are notified in FIFO order. Memory client device drivers are clients that use the **Open/ Close/ Read/ Write/ Copy/ EraseMemory** requests and so must be notified after the MTDs have associated with the memory regions that they will support. IO clients must use the **RequestConfiguration** function to set the socket and card interface type before MTDs and memory clients begin accessing the card.

5.32 RegisterEraseQueue (0FH)

```
CardServices  (RegisterEraseQueue, ClientHandle/EraseQueueHandle,
               EraseQueueHeader, 0, null)
```

This function registers a client supplied erase queue with Card Services. The pointer to the EraseQueueHeader is passed in the Pointer argument. The ClientHandle returned by **RegisterClient** is passed in the Handle argument. The EraseQueueHandle for the Queue is returned in the Handle argument.

The **EraseQueueHeader** has the following structure:

Offset	Field	Size	Type	Value	Detail/Description
0	QueueEntryLen	2	I	N	Length in bytes of an erase queue entry.
2	QueueEntryCnt	2	I	N	Number of entries in the erase queue.
4	QueueEntryArray	N	I	N	Array of Erase Queue Entries

The **QueueEntryLen** field specifies the size in bytes of each entry in the erase queue. The erase queue entries start at offset 4 from the beginning of the **EraseQueueHeader**, i.e. immediately after the **QueueEntryCnt** field of the erase queue **EraseQueueHeader**. The erase queue entries are contiguous and form an array of entries.

The **QueueEntryCnt** field specifies the number of entries in the erase queue.

Each **QueueEntry** has the following structure:

Offset	Field	Size	Type	Value	Detail/Description
0	Handle	2	I	N	MemoryHandle.
2	EntryState	1	I/O	N	State of this erase queue entry
3	Size	1	I	N	Size of area to be erased (power of 2)
4	Offset	4	I	N	Offset of area to be erased
8	Optional	N	I	N	Additional bytes for client use.

The **Handle** field contains the memory handle returned by an **OpenMemory** request for the memory area to be erased by this erase request.

The **EntryState** field indicates the state of this queue entry. The following states are defined:

State	Description
IDLE (FFH)	The erase queue entry has no valid information and should be ignored by Card Services. Should only be set when Card Services is not processing the request (as indicated by a value other than 01H-7FH).
QUEUED_FOR_ERASE (00H)	The client has queued this entry for erasure. The client must not modify this entry until Card Services indicates that the erase has been processed. Only set by the Client before a **RegisterEraseQueue** or **CheckEraseQueue** request.
ERASE_IN_PROGRESS (01H-7FH)	Card Services has started processing this entry. Only set by Card Services when the request is being serviced.
ERASE_PASSED (E0H)	Card Services has completed processing this entry and the erase was successful. The client can modify this entry. Only set by Card Services when the request has been serviced.
ERASE_FAILED (E1H)	Card Services has completed processing this entry and the erase was unsuccessful. The client can modify this entry. This entry code is only set by Card Services when the indicated block could not be erased after appropriate retries. The client is expected to treat the block indicated by this entry as unusable. Only set by Card Services when the request has been serviced.
MEDIA_WRITE_PROTECTED (84H) MEDIA_NOT_ERASABLE (86H) MEDIA_MISSING (80H) MEDIA_NOT_WRITABLE (87H)	Card Services has completed processing this entry and the erase was unsuccessful. The client can modify this entry. These codes indicate a user induced failure that can be corrected with appropriate user interaction. Only set by Card Services when the request cannot be serviced. MEDIA_NOT_ERASABLE is returned when an erase is attempted on an SRAM card supported by the default Card Services SRAM MTD.
BAD_SOCKET (C3H) BAD_TECHNOLOGY (C2H) BAD_OFFSET (C1H) BAD_VCC (C4H) BAD_VPP (C5H) BAD_SIZE (C6H)	Card Services has completed processing this entry and the erase was not attempted. The client can modify this entry. These codes indicate an error in the parameters of the entry. Only set by Card Services when the request cannot be serviced.

The **Size** field specifies the size of the memory area to be erased. It must be the exponent for a power of 2, e.g. a 64KB erase block request would specify a size of 16.

The **Offset** field specifies the offset to the memory area to be erased. It is a relative offset from the physical offset provided to **Open Memory** when the **MemoryHandle** was obtained. The offset must be the beginning of an erase block.

The **Optional** field is a byte array that can be used by the client and will not be accessed by Card Services.

All entries in an erase queue may be idle when registered. A return code of SUCCESS indicates the erase queue will be serviced by Card Services.

See also **DeregisterEraseQueue**.

Return Codes:

BAD_ARGS	QueueEntryCnt less than 1 or QueueEntryLen less than 6
BAD_ARG_LENGTH	*ArgLength is not equal to zero (0)*
BAD_HANDLE	ClientHandle is invalid

5.33 RegisterMTD (1AH)

```
CardServices(RegisterMTD, ClientHandle, null, ArgLength, ArgPointer)
```

This function allows a Memory Technology Driver to register to handle accesses for a region by a memory function. The ClientHandle returned by **RegisterClient** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Attributes of memory type.
4	Offset	4	I	N	Card Offset for Region MTD supports
8	MTD Media ID	2	I	N	Token for MTD use to identify media

The **Socket** identifies the logical socket containing the desired card.

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

Bit 0	Memory type (set = attribute)
Bits 1 -- 8	RESERVED (Reset to zero)
Bits 9 -- 10	Write/Erase interactions: 0 - Write without Erase 1 - Write with Erase 2 - Reserved 3 - Write with Disableable Erase
Bit 11	Write with Verify
Bit 12	Erase Requests Supported
Bits 13 -- 15	RESERVED (Reset to zerb)

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

The **Offset** field identifies the memory region for which the MTD supports access. This value must be the beginning address of a region.

The **MTD Media ID** is a value that Card Services maintains on behalf of an MTD. The MTD uses this value to indicate the type of memory media for this region. This value is passed to the MTD by Card Services whenever a read, write, or erase memory access function is requested for this region.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to ten (10)
BAD_HANDLE	ClientHandle is invalid
BAD_OFFSET	Offset is invalid
BAD_SOCKET	Socket is invalid
NO_CARD	No PC Card in socket

5.34 RegisterTimer (28H)

```
CardServices(RegisterTimer, ClientHandle/TimerHandle, null, ArgLength,
             ArgPointer)
```

This function registers a callback structure with Card Services. Based on a tick count provided, Card Services calls the client back when the time period has elapsed and the Card Services interface is available. The ClientHandle returned by RegisterClient is passed in the Handle argument. A TimerHandle is returned in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Wait	2	I	N	Number of ticks to wait

The Wait field is the number of timer intervals Card Services should wait before notifying the client. The tick interval is approximately 1 ms. See warning below. If the Wait field is zero (0) on entry, Card Services notifies the client as soon as the Card Services interface is present.

This function is intended for use by two types of clients. First, those who may be operating in a background thread of execution. Second, clients who need periodic wakeup calls.

Background thread of execution clients may require Card Services to perform a task when the Card Services interface is busy with a foreground request. Since this function is always available, even when the Card Services interface reports BUSY on all other requests, it allows a client to schedule a later wakeup when Card Services is available to perform the delayed request.

This function may also serve as a method of receiving periodic wakeup notifications without a client having to intercept the system timer tick. As noted above, a side effect is that when the client receives notification the timer has expired, the Card Services interface is guaranteed to be available.

When the timer expires and the Card Services interface is available, Card Services notifies the client at the address specified when the client was registered with RegisterClient. Clients can use the TimerHandle passed to their callback routine to determine which timer has expired.

A client may have more than one timer pending at a time Card Services returns OUT_OF_RESOURCE if it cannot accept the timer request.

WARNING:

When Card Services is operating in some environments, timer intervals may not be received by Card Services reliably. Many environments (such as Microsoft Windows or systems running LIM emulators on Virtual-86 capable processors) may generate timer intervals in bursts. The actual interval available on a system may not be as little as 1 ms. This interval should only be viewed as advisory. Other Operating System specific services should be used for reliable timing.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to two (2)
BAD_HANDLE	ClientHandle is invalid
OUT_OF_RESOURCE	Timer request invalid

5.35 ReleaseConfiguration (1EH)

```
CardServices(ReleaseConfiguration,   ClientHandle,   null,   ArgLength,
                ArgPointer)
```

This function returns a PC Card and socket to a simple memory only interface and configuration zero. Card Services may remove power from the socket if no clients have indicated their usage of the socket by an **OpenMemory** *or* ***RequestWindow****. Card Services is prohibited from resetting the PC Card and is not required to cycle power through zero (0) volts.* The ClientHandle used in **RequestConfiguration** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket

The **Socket** identifies the logical socket to release.

BAD_HANDLE is returned if the ClientHandle is not the one passed to **RequestConfiguration**.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to two (2)
BAD_HANDLE	ClientHandle does not match owning client *or no configuration to release.*
BAD_SOCKET	Socket is invalid

5.36 ReleaseExclusive (2DH)

```
CardServices(ReleaseExclusive, ClientHandle, null, ArgLength, ArgPointer)
```

This function releases the exclusive use of a card in a socket for a client. The ClientHandle passed to **RequestExclusive** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit Mapped field.

The **Socket** field identifies the logical socket that contains the desired PC Card.

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

Bits 0 -- 15	RESERVED (Reset to zero)

Card Services returns to the client immediately. As soon as the Card Services interface is available, Card Services sends a CARD_REMOVAL event to the requesting client and then sends a CARD_INSERTION event to all registered clients.

See also the **RequestExclusive** function.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to four (4)
BAD_HANDLE	ClientHandle does not match owning client *or no client has exclusive use of the PC Card in the socket.*
BAD_SOCKET	Socket is invalid

5.37 ReleaseIO (1BH)

`CardServices(ReleaseIO, ClientHandle, null, ArgLength, ArgPointer)`

This function releases previously requested I/O addresses. Only the Card Services database of resources is adjusted by this function. No changes are made to the socket adapter. **ReleaseIO** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket without a matching **ReleaseConfiguration**. The ClientHandle used in Request IO is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Base Port1	2	I	N	Base port address for range 1
4	Num Ports1	1	I	N	Number of contiguous ports
5	Attributes1	1	I	N	Bit-mapped
6	Base Port2	2	I	N	Base port address for range 2
8	Num Ports2	1	I	N	Number of contiguous ports
9	Attributes2	1	I	N	Bit-mapped
10	IOAddrLines	1	I	N	Number of IO address lines decoded by PC Card

The **Socket** field identifies the logical socket containing the PC Card.

The **Base Port** fields describe the first port address assigned by a **RequestIO** request. It should match the value returned by the **RequestIO** function exactly.

The **Num Ports** fields describe the number of contiguous ports assigned by a **RequestIO** function.

The **Attributes** field is defined the same as for **RequestIO** and must have the same value returned by **RequestIO**.

The **IOAddrLines** field is the number of IO address lines decoded by the PC Card in the specified socket. It is used by Card Services to determine whether any addresses outside the ranges specified needed to be marked as in-use because the combination of socket hardware and lines decoded could result in PC Card accesses outside the specified ranges.

Releasing ports using different **Base Port, Num Ports** or **Attributes** values then those used by the corresponding **RequestIO** is not supported. Doing so may crash the host system or confuse resource allocation.

SUCCESS is returned if the specified ports were in use and have been released.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to eleven (11)
BAD_ARGS	I/O description doesn't match allocation
BAD_HANDLE	ClientHandle does not match owning client *or no I/O ports to release.*
BAD_SOCKET	Socket is invalid
CONFIGURATION_LOCKED	Configuration has not been released

5.38 ReleaseIRQ (1CH)

```
CardServices(ReleaseIRQ, ClientHandle, null, ArgLength, ArgPointer)
```

This function releases a previously requested interrupt request line. Only the Card Services database of resources is adjusted by this function. No changes are made to the socket adapter. **ReleaseIRQ** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket without a matching **ReleaseConfiguration**. The ClientHandle used in **RequestIRQ** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped
4	AssignedIRQ	1	I	N	IRQ Number Assigned by CS

The **Socket** field identifies the logical socket to disable IRQ steering.

The **Attributes** field is bit-mapped and is defined the same as in **RequestIRQ**.The **Assigned IRQ** field identifies the IRQ that was previously established by **RequestIRQ**.

> **Note:**
> **Most systems have hardware support for interrupt reporting and an interrupt handler vector (jump) table. This function does not manipulate any such motherboard specific hardware nor does it manipulate the vector table. It is up to the client to perform these activities, if the IRQ is not shared before invoking this function. No adjustment of the interrupt vectors in the interrupt table is made by this request. It is up to the client to restore the original interrupt handler after invoking this function.**

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to five (5)
BAD_ATTRIBUTE	Attributes don't match allocation
BAD_IRQ	AssignedIRQ doesn't match allocation
BAD_HANDLE	ClientHandle does not match owning client *or no IRQ to release.*
BAD_SOCKET	Socket is invalid
CONFIGURATION_LOCKED	Configuration has not been released

5.39 ReleaseSocketMask (2FH)

```
CardServices(ReleaseSocketMask, ClientHandle, null, ArgLength, ArgPointer)
```

This function requests that the client no longer be notified of status changes for this socket. A client will still be notified of status changes for this socket if it has events enabled in its global event mask. The ClientHandle passed to **RequestSocketMask** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket

The **Socket** field describes the logical socket to stop notifying the client of status changes.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to two (2)
BAD_HANDLE	ClientHandle does not match owning client *or no socket mask to release.*
BAD_SOCKET	Socket is invalid

5.40 ReleaseWindow (1DH)

```
CardServices(ReleaseWindow, WindowHandle, null, 0, null)
```

This function releases a block of system memory space which was obtained previously by a corresponding **RequestWindow**. The WindowHandle returned by **RequestWindow** is passed in the Handle argument.

Card Services assumes only the owning client will make this request. For that reason, the owning client handle is not provided as an argument. Card Services maintains the owning client's handle in its internal database to properly manage resources.

Return Codes:

BAD_ARG_LENGTH	*ArgLength is not equal to zero (0)*
BAD_HANDLE	WindowHandle is invalid

5.42 RequestConfiguration (30H)

```
CardServices(RequestConfiguration,   ClientHandle,   null,   ArgLength,
              ArgPointer)
```

This function configures the PC Card and socket. Card Services will apply power to the socket if the socket was not powered. This function must be used by clients that require an I/O interface to their PC Card. The ClientHandle returned by **RegisterClient** is passed in the Handle argument.

RequestIO and **RequestIRQ** must be used before calling this function to specify the IO and IRQ requirements for the PC Card and socket. After finding an appropriate configuration, **RequestConfiguration** establishes the configuration in the socket adapter and PC Card.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Vcc	1	I	N	Vcc Setting
5	Vpp1	1	I	N	Vpp1 Setting
6	Vpp2	1	I	N	Vpp2 Setting
7	IntType	1	I	N	Memory or Memory+I/O Interface
8	ConfigBase	4	I	N	Card Base address of config registers
12	Status	1	I	N	Card Status register setting, if present
13	Pin	1	I	N	Card Pin register setting, if present
14	Copy	1	I	N	Card Socket/Copy register setting, if present
15	ConfigIndex	1	I	N	Card Option register setting, if present
16	Present	1	I	N	Card Configuration registers present

The **Socket** field identifies the logical socket for which to set the configuration.

The **Attributes** field is bit-mapped. It indicates whether the client wishes exclusive use of the socket. The routing of the IRQ can also be enabled. The following bits are defined:

Bit 0	RESERVED (Reset to zero)
Bit 1	Enable IRQ steering (set = true)
Bits 2 -- 15	RESERVED (Reset to zero)

Setting Bit 1 to one enables the IRQ steering as requested by **RequestIRQ**. Resetting Bit 1 to zero disables the IRQ steering.

The **Vcc, Vpp1** and **Vpp2** fields all represent voltages expressed in tenths of a volt. Since these fields are a byte wide, values from zero (0) to 25.5 volts may be set. To be valid, the exact voltage must be available through the system's Socket Services. To be compliant with the *PCMCIA PC Card Standard - Release 2.0*, systems must always support 5.0 volts for both Vcc and Vpp.

The **ConfigBase** field is the offset in attribute memory of the configuration registers. The **Present** field identifies which, if any, of the configuration registers are present. If present, the corresponding bit is set. This field is bit-mapped as follows:

Bit 0	Option
Bit 1	Status
Bit 2	Pin Replacement
Bit 3	Copy
Bits 4 -- 7	RESERVED (Reset to zero)

The **IntType** field is bit-mapped. It indicates how the socket should be configured. The following bits are defined:

Bit 0	Memory (set = true)
Bit 1	Memory and I/O (set = true)
Bits 2 -- 7	RESERVED (Reset to zero).

Only one client can be in control of the interface type at any time. Once an interface type other than Memory has been set, a **ReleaseConfiguration** must be used to return the socket to a memory only interface before another IO interface configuration can be selected.

The **Status**, **Pin** and **Copy** fields represent the initial values *that* ~~which~~ should be written to those registers if they are present, *as indicated by the* ***Present*** *field. The* ***Pin*** *field is also used to inform Card Services which pins in the PC Card's Pin Replacement Register are valid, if any. Only those bits which are set are considered valid. This affects how status is returned by the* ***GetStatus*** *function. If a particular signal is valid in the Pin Replacement Register, both the mask bit and the change bit must be set in the* ***Pin*** *field.*

The **ConfigIndex** field is the value written to the Option register for the configuration index required by the PC Card. *Only the least significant six bits are significant, the upper two (2) bits are ignored. The interrupt type is set by Card Services based on the client's prior Request IRQ () request and the host hardware environment.*

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to seventeen (17)
BAD_ATTRIBUTE	IRQ steering enable conflict
BAD_HANDLE	ClientHandle is invalid
BAD_TYPE	I/O and memory interface not supported
BAD_SOCKET	Socket is invalid
BAD_VCC	Requested Vcc is not available on socket
BAD_VPP	Requested Vpp is not available on socket
CONFIGURATION_LOCKED	Configuration already set
NO_CARD	No PC Card in socket
IN_USE	PC Card already being used

5.43 RequestExclusive (2CH)

```
CardServices(RequestExclusive, ClientHandle, null, ArgLength, ArgPointer)
```

This function requests the exclusive use of a PC Card in a socket for a client. Clients currently using the PC Card in the socket can reject the request. The ClientHandle returned by **RegisterClient** is passed in the Handle argument. This function returns without indicating whether the client received exclusive access to the PC Card. The client is notified via its callback entry point whether it has received exclusive access. This notification can happen before or after this function returns.

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

The **Socket** field identifies the logical socket that contains the desired PC Card.

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

Bits 0 -- 15	RESERVED (Reset to zero)

Card Services returns to the client immediately after noting the request . As soon as the Card Services interface is available, Card Services sends EXCLUSIVE_REQUEST events to registered clients for this socket. If any client returns failure for the EXCLUSIVE_REQUEST event, Card Services terminates notification processing. Card Services then notifies the requesting client that the exclusive request failed via an EXCLUSIVE_COMPLETE event with the Info argument set to the return code set by the client that rejected the request.

Once all clients have accepted the EXCLUSIVE_REQUEST event, Card Services sends CARD_REMOVAL events to all clients registered and then sends a CARD_INSERTION event to the requesting client. Finally, Card Services sends the EXCLUSIVE_COMPLETE event to the requesting client.

When the client is through using the PC Card in an exclusive fashion, it must use the **ReleaseExclusive** function to return the PC Card to normal use.

Return Codes:

BAD_ARG_LENGTH	ArgLength not equal to four (4)
BAD_HANDLE	ClientHandle is invalid
BAD_SOCKET	Socket is invalid
IN_USE	PC Card already in-use exclusively
NO_CARD	No PC Card in socket

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

Bit 0	Shared (set = true)
Bit 1	First Shared (set = true)
Bit 2	Force Alias Accessibility
Bit 3	Data Path Width for I/O Range (reset to zero = 8 bit, set to one = 16 bit)
Bits 4 -- 7	RESERVED (Reset to zero)

5.44 RequestIO (1FH)

```
CardServices(RequestIO, ClientHandle, null, ArgLength, ArgPointer)
```

This function requests I/O addresses for a socket. **RequestIO** and **RequestIRQ** must be called to specify how a PC Card and socket configuration should be set. If a specific combination of IO and IRQ configuration is required, **ReleaseIO/IRQ** may need to be used to release a partial configuration if the complete configuration cannot be achieved. Then **RequestConfiguration** must be used to establish the configuration. **RequestIO** and **RequestIRQ** can be requested multiple times until a successful complete configuration is found. **RequestConfiguration** only uses the last configuration specified. **RequestIO** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket. **RequestIO** fails if it has already been called without a corresponding **ReleaseIO**. The ClientHandle *of the requesting client* ~~returned by **RegisterClient**~~ is passed in the Handle argument.

> **Note:**
> RequestIO **should be called** ***once*** **per socket.**

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Base Port1	2	I/O	N	Base port address for range
4	Num Ports1	1	I	N	Number of contiguous ports
5	Attributes1	1	I	N	Bit-mapped
6	Base Port2	2	I/O	N	Base port address for range
8	Num Ports2	1	I	N	Number of contiguous ports
9	Attributes2	1	I	N	Bit-mapped
10	IOAddrLines	1	I	N	Number of IO address lines decoded

The **Socket** field identifies the logical socket that contains the desired PC Card.

Two I/O address ranges can be requested by **RequestIO**. Each I/O address range is specified by the **BasePort**, **NumPorts**, and **Attributes** fields. If only a single I/O range is being requested, the NumPorts2 field must be reset to zero.

The **Base Port** field is the first port address requested. If reset to zero (0), Card Services returns an I/O address based on the available I/O addresses and the number of contiguous ports requested. *When **Base Port** is zero, Card Services aligns the returned range in the host system's I/O address space on a boundary that is a multiple of the number of contiguous ports requested rounded up to the nearest power of two. For example, if a client requests two I/O ports, the returned **Base Port** value will be a multiple of two. If a client requests five contiguous I/O ports, the returned **Base Port** value will be a multiple of eight.* If multiple ranges are being requested, the **Base Port** field must be non-zero for all specified ranges.

The **Num Ports** field is the number of contiguous ports being requested.

Normally, each request dedicates the requested ports to the indicated socket, if they are available. However, for some applications and/or cards, ports may be shared by cards in two or more sockets. If the **Shared** bit is set, the ports requested may be shared with another socket. **First Shared** is *additionally* set when a previously unshared I/O range is required that is intended to be shared with subsequent clients using this same I/O range. If a previously unshared I/O range is unavailable, the function fails and IN_USE is returned.

If a shared I/O range is requested, *the client is responsible for determining whether the range may be shared* ~~the ClientHandle of the first sharing client (which may not be the handle for this client) must be passed to this request~~.

A PC Card may decode less than the full set of possible I/O address lines. Doing so creates aliased addresses for the PC Card address range by using different values for the undecoded upper address lines. **Force Alias Accessibility** requests that the aliased address ranges be configured so that they can also be used to address the PC Card. This is used for compatibility with similar functionality on existing ISA bus style add-in cards. If Card Services can only satisfy this request by aliasing the requested IO addresses with other IO addresses and **Force Alias** is not set to one, the BAD_ATTRIBUTE error will be returned.

The **Data Path Width for I/O Range** is reset to zero to indicate that the I/O addresses are 8 bit registers. The field is set to one for 16 bit registers.

> **NOTE:**
> **This capability may not be available for all systems and clients should not depend on being able to request an address range with alias accessibility.**

Shared ports are managed internally by Card Services with share counts. Once internal share counts reach zero (0), those ports may be reassigned for exclusive use. If share counts for I/O ports are non-zero, they may only be shared if requested by **RequestIO**. If socket hardware does not support shared I/O ports for a shared request or is unable to satisfy a non-shared request, this function returns failure.

The **IOAddrLines** field is the number of I/O address lines decoded by the PC Card in the specified socket. It is used by Card Services to determine whether any addresses outside the ranges specified needed to be marked as in-use because the combination of socket hardware and lines decoded could result in PC Card accesses outside the specified ranges.

SUCCESS is returned if the specified ports are available.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to eleven (11)
BAD_ATTRIBUTE	Sharing or alias request invalid
BAD_BASE	Base port address is invalid
BAD_HANDLE	ClientHandle is invalid
BAD_SOCKET	Socket is invalid
CONFIGURATION_LOCKED	Configuration already set
IN_USE	I/O ports requested are already in-use *or function has already been successfully invoked*
NO_CARD	No PC Card in socket
OUT_OF_RESOURCE	Internal data space exhausted

5.45 RequestIRQ (20H)

```
CardServices(RequestIRQ, ClientHandle, null, ArgLength, ArgPointer)
```

This function requests an interrupt request line. **RequestIO** and **RequestIRQ** must be called to specify how a PC Card and socket configuration should be set. If a specific combination of IO and IRQ configuraiton is required, **ReleaseIO/IRQ** may need to be used to release a partial configuration if the complete configuration cannot be achieved. Then **RequestConfiguration** must be used to establish the configuration. **RequestIO** and **RequestIRQ** can be requested multiple times until a successful complete configuration is found. **RequestConfiguration** only uses the last configuration specified. **RequestIRQ** returns error code CONFIGURATION_LOCKED if **RequestConfiguration** has already been used for this socket. The ClientHandle returned by **RegisterClient** is passed in the Handle argument.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Bit-mapped (defined below)
4	AssignedIRQ	1	O	N	IRQ Number Assigned by CS
5	IRQInfo1	1	I	N	First PCMCIA IRQ Byte
6	IRQInfo2	2	I	N	Optional PCMCIA IRQ bytes

The **Socket** field identifies the logical socket containing the PC Card desired.

The **Attributes** field is bit-mapped. It specifies details about the type of IRQ desired by the client.

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

Bits ·· 1	IRQ type: 0 - Exclusive 1 - Time-Multiplexed Sharing 2 - Dynamic Sharing 3 - RESERVED (Reset to zero)
Bit 2	Force Pulse (set = true)
Bit 3	First Shared (set = true)
Bit 4 ·· 7	RESERVED (Reset to zero)
Bit 8	Pulse IRQ Allocated (set = true on return)
Bits 9 ·· 15	RESERVED (Reset to zero)

The **IRQ** type field specifies the characteristics of the IRQ requested by the client. **Exclusive** indicates that the system IRQ is dedicated to this PC Card.

Time-Multiplexed Sharing indicates the client shares the system IRQ with other PC Cards. This client coordinates with other clients in using **ModifyConfiguration** to enable/disable the IRQ from each socket. This ensures that only one PC Card is connected to the system IRQ line at any time. A time-multiplexed IRQ is only supported for interrupts that can be enabled and disabled at the socket.

Dynamic Sharing indicates that this PC Card will share the system IRQ simultaneously with other PC Cards. The clients of the PC Cards use card features to identify the source of the interrupt. Dynamic sharing is available via PC Card level interrupts in systems that support level mode interrupts. Dynamic sharing is also available via PC Card pulse interrupts in systems that support pulse mode interrupts. Level mode interrupts are assigned if possible. **ForcePulse** is used to force Card Services to use a pulse mode interrupt.

First Shared is set when a previously unshared IRQ is required that is intended to be shared with subsequent clients using this same IRQ. If a previously unshared IRQ is unavailable, the function fails and BAD_IRQ is returned. **First Shared** is only valid when **Time Multiplexed Sharing** or **Dynamic Sharing** is specified for the IRQ type.

Pulse IRQ Allocated is valid only for dynamic shared IRQs. It indicates whether the PC Card will be configured for pulse or level operation.If a shared IRQ is requested, the ClientHandle of the *requesting* ~~first sharing~~ client must be used for this request.

Card Services maintains internal share counts for returned IRQ values that indicate sharing. These counts are incremented as IRQ are assigned and decremented when released. If an IRQ internal share count is zero, the IRQ may be exclusively assigned to a socket.

The **AssignedIRQ** field is returned by Card Services if one of the requested IRQ levels is available and was assigned to the socket. It is a value from zero (0) to nineteen (19). Zero (0) through fifteen (15) correspond to IRQ levels 0 through 15, respectively.

The **IRQInfo1** field is bit-mapped. It indicates the capabilities of the PC Card in the socket. Its structure is identical to the first PCMCIA Interrupt Description IRQ byte. The structure is:

if Bit 4 is *reset* to *zero* ~~one~~:

Bits 0 - 3	IRQN if Bit 4 is *reset* to *zero* ~~one~~

if Bit 4 is ~~reset~~ to *one* ~~zero~~:

Bit 0	NMI
Bit 1	IOCK
Bit 2	BERR
Bit 3	VEND

Bit 4	Mask, If set to one, the IRQInfo2 bytes are valid

Bit 5	Level
Bit 6	Pulse
Bit 7	Share

The **IRQInfo2** field is bit-mapped. These two bytes have identical structure to the PCMCIA Interrupt Description optional IRQ bytes. The structure is:

Bit 0	IRQ0
Bit 1	IRQ1
Bit 2	IRQ2
Bit 3	IRQ3
Bit 4	IRQ4
Bit 5	IRQ5
Bit 6	IRQ6
Bit 7	IRQ7

Bit 8	IRQ8
Bit 9	IRQ9
Bit 10	IRQ10
Bit 11	IRQ11
Bit 12	IRQ12
Bit 13	IRQ13
Bit 14	IRQ14
Bit 15	IRQ15

Multiple bits may be set in the **IRQInfo1** and **IRQInfo2** fields. At least one bit must be set in these fields. How Card Services selects one level over another when more than one level is currently available is implementation-specific.

> **Note:**
> **Most systems have hardware support for interrupt reporting and an interrupt handler vector (jump) table. This function does not manipulate any such motherboard specific hardware nor does it manipulate the vector table. It is up to the client to perform these activities, if the IRQ is not shared before invoking this function. No adjustment of the interrupt vector in the interrupt table is made by this request. It is up to the client to set the interrupt handler before invoking this function.**

Clients MUST ensure that the PC Card does not generate an interrupt before the interrupt handler is installed by the client on the appropriate interrupt vector.

SUCCESS is returned if one of the specified IRQs is available with the sharing requested. BAD_IRQ is returned if no IRQ satisfying the request can be found.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to eight (8)
BAD_ARGS	IRQ Info fields are invalid
BAD_ATTRIBUTE	Sharing or pulse request invalid
BAD_HANDLE	ClientHandle is invalid
BAD_IRQ	IRQ is invalid (returned by Socket Services)
BAD_SOCKET	Socket is invalid
CONFIGURATION_LOCKED	Configuration already set
IN_USE	IRQ requested is already in-use *or function has already been successfully invoked*
NO_CARD	No PC Card in socket

5.46 RequestSocketMask (22H)

`CardServices(RequestSocketMask, ClientHandle, null, ArgLength, ArgPointer)`

This function requests that the client be notified of status changes for this socket. If the client also has events enabled in its global event mask, it may be notified more than once for each status change for this socket. The ClientHandle is passed in the Handle argument. **RequestSocketMask** must be used before a **Get/SetEventMask** request for this socket will succeed.

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

The Socket field identifies the logical socket for which events should be enabled.

The Event Mask is a bit-mapped field. It represents the status callback events to notify the client of when they occur on this socket. The bits are defined as follows:

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

BAD_SOCKET is returned if the Socket field is invalid.

> *Note:*
> *Socket event masks must be released when a CARD_REMOVAL notification is received.*

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to four (4)
BAD_HANDLE	ClientHandle is invalid
BAD_SOCKET	Socket is invalid
NO_CARD	No PC Card in socket

5.47 RequestWindow (21H)

```
CardServices(RequestWindow, ClientHandle/WindowHandle, null, ArgLength,
                ArgPointer)
```

This function requests a block of system memory space be assigned to a memory region of a PC Card in a socket. The ClientHandle *of the requesting client* ~~returned by RegisterClient~~ is passed in the Handle argument. The WindowHandle is returned in the Handle argument. This WindowHandle must be passed to ReleaseWindow when the client is done using the window.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I/O	N	Memory Window Attribute Field
4	Base	4	I/O	N	System Base Address
8	Size	4	I/O	N	Memory Window Size
12	AccessSpeed	1	I	N	Window Speed Field

The Socket field describes the logical socket to map into system memory address space.

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

Bit 0	RESERVED (Reset to 0)
Bit 1	Memory type (set = attribute)
Bit 2	Enabled (set = true, reset = disabled)
Bit 3	Data path width (reset = 8-bit/set = 16-bit)
Bit 4	Paged (set = true)
Bit 5	Shared (set = true)
Bit 6	First Shared (set = true)
Bit 7	*Binding Specific*~~RESERVED (reset to zero)~~
Bit 8	Card offsets are window sized (set = true)
Bits 9 -- 15	RESERVED (reset to zero)

The Enabled bit indicates whether the window hardware should be enabled. A client may allocate multiple windows into the same system space as long as the client time-multiplexes them. Only one PC Card may respond to accesses into the shared system space at a time or hardware damage may result.

If the Paged bit is set to one, the window size must be a multiple of 16 kbytes. The first 16 kbytes of system memory address space used to map PC Card memory into system memory is referred to as page zero (0). The next 16 kbytes is page one (1), and so on. A 48 kbyte window has three (3) pages numbered 0, 1, and 2. If the Paged bit is reset to zero, the window size is determined by the Size field. In both cases, the PC Card memory offset for the window is set by MapMemPage.

Normally, each request dedicates the requested system memory address range to the indicated socket, if it is available. However, for some applications and/or cards, system memory address space may be shared by cards in two or more sockets. If the Shared bit is set, the memory range requested may be shared with another socket. First Shared is set when a previously unshared system memory range is required that is intended to be shared with subsequent clients using this same system memory range. If a previously unshared system memory range is unavailable, the function fails and OUT_OF_RESOURCE is returned. The **Shared** bit may be used to allocate multiple windows mapping into the same system address space. This prevents Card Services from failing the request because the system address space has already been allocated to another window that was requested with **Shared** set to one. PC Cards must ensure that multiple cards mapped to the same system address don't all respond to memory accesses.

Card offsets are window sized is set by Card Services when the client must specify card offsets that are a multiple of the window size.

The **Base** field points to the physical location in system address space to map card memory. If reset to zero (0) on entry, Card Services attempts to locate an available area of system address space. If successful, Card Services returns the base system address in this field. The **Size** field is the byte size of the memory window requested. **Size** may be zero to indicate that that Card Services should provide the largest size available.

Size and **Base** are in bytes, but must be of a supported granularity and alignment. If a client intends to map multiple windows into the same system memory, the first request should allow Card Services to determine the window base. Subsequent requests for windows using the same system memory MUST specify the base address returned by the first request. All windows should use the same size value. Only one request may set the Enabled bit in the **Attribute** field.

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

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

The above bit definitions use the format of the extended speed byte of the Device ID tuple. If the mantissa is zero (noted as reserved in the *PCMCIA PC Card Standard - Release 2.0*), the lower bits are a binary code representing a speed from the table below:

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

The **MapMemPage** and **ModifyWindow** functions use the values returned by this function.

SUCCESS is returned if a window has been defined.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to thirteen (13)
BAD_ATTRIBUTE	Data width or paging invalid or sharing conflicts
BAD_BASE	System memory address invalid
BAD_HANDLE	ClientHandle is invalid
BAD_SIZE	Window size is invalid
BAD_SOCKET	Socket is invalid
BAD_SPEED	Speed not supported
NO_CARD	No PC Card in socket
OUT_OF_RESOURCE	Internal data space is exhausted
IN_USE	Window requested is in use

5.48 ResetCard (11H)

```
CardServices(ResetCard, ClientHandle, null, ArgLength, ArgPointer)
```

This function resets the PC Card in the specified socket. The ClientHandle returned by **RegisterClient** is passed in the Handle argument. The actual reset card processing is done in the background asynchronously from the execution of this function. The client receives a RESET_COMPLETE upon completion of this processing.

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

The **Socket** field identifies the logical socket that contains the PC Card to be reset.

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

Bits 0 -- 15	RESERVED (Reset to zero).

Card Services returns to the client after noting the request. As soon as the Card Services interface is available, Card Services sends RESET_REQUEST events. If any client rejects the RESET_REQUEST, Card Services terminates reset processing. Card Services then notifies the requesting client via a RESET_COMPLETE event with the Info argument set to the return code set by the client that rejected the request.

If no client rejects the reset request, Card Services sends a RESET_PHYSICAL to all clients that have indicated their interest in Reset events so they may prepare for a hardware reset. When all clients have been notified, Card Services performs a hardware reset. Card Services observes the appropriate wait after reset hardware timing including monitoring the RDY/BSY signal from the PC card. If a BSY/RDY transition occurs due to the reset, Card Services doesn't generate a CARD_READY event. Then Card Services sends a CARD_RESET notification to all registered clients.

Finally, Card Services notifies the requesting client's callback handler of a RESET_COMPLETE event. When control is returned from the background thread Card Services has been performing, the requesting client may continue processing.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to four (4)
BAD_HANDLE	ClientHandle does not match owning client
BAD_SOCKET	Socket is invalid
NO_CARD	No PC Card in socket

5.49 ReturnSSEntry (23H)

`CardServices(ReturnSSEntry, null, SSentry, ArgLength, ArgPointer)`

This function returns a pointer to an entry point that can be used to call Socket Services. The entry point is returned in the Pointer argument. The entry point references code in Card Services that calls the correct Socket Services entry point as required based on the physical adapter and socket numbers provided by the Socket Services client.

Offset	Field	Size	Type	Value	Detail/Description
0	Attributes	2	I	N	Information about the SS entry point

The **Attributes** field specifies details about the Socket Services entry point. It is defined the same as the **Attributes** field in **Add/ReplaceSocketServices.**

WARNING:

Direct access to Socket Services functions which modify hardware state may cause the host system to crash. Clients should limit their access to functions which only return state information.

Note:
Making a request of Socket Services through this function is considered the same as a request through the Card Services interface. Subsequent Socket and Card Services requests are blocked until the prior request is complete. Card Services monitors requests made through this entry point to avoid performing asynchronous callback notifications when the interfaces are not available.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to two (2)
UNSUPPORTED_FUNCTION	Implementation does not support function
UNSUPPORTED_MODE	Processor mode not supported

5.50 SetEventMask (31H)

```
CardServices(SetEventMask, ClientHandle, null, ArgLength, ArgPointer)
```

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

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

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

Bit 0	Event mask of this socket only (set = true)
Bits 1 - 15	RESERVED (Reset to zero)

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

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

Bit 0	Write Protect
Bit 1	Card Lock Change
Bit 2	Ejection Request
Bit 3	Insertion Request
Bit 4	Battery Dead
Bit 5	Battery Low
Bit 6	Ready Change
Bit 7	Card Detect Change
Bit 8	PM Change
Bit 9	Reset
Bit 10	SS Update
Bits11 - 15	RESERVED (reset to zero)

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

The **Socket** identifies the logical socket when **Attributes** Bit 0 is set.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to six (6)
BAD_HANDLE	ClientHandle is invalid
BAD_SOCKET	Socket is invalid (socket requests, only) or this socket has not been requested via RequestSocketMask
NO_CARD	No PC Card in socket (socket requests, only)

[illegible] SetRegion (29H)

```
CardServices(SetRegion, null, null, ArgLength, ArgPointer)
```

This function allows a client to set a PC Card region's characteristics. It is intended to allow region characteristics to be set when they are not available in the Card Information Structure.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Attributes	2	I	N	Bit-mapped (defined below)
4	Card Offset	4	I	N	Card Memory Region Offset
8	Region Size	4	I	N	Region Size
12	EffBlockSize	4	I	N	Erase Block Size
16	PartMultiple	2	I	N	Partition Multiple (Erase Block units)
18	JEDEC ID	2	I	N	Partition JEDEC Memory ID Code
20	Bias Offset	4	I	N	Address Bias for MTD
24	Access Speed	1	I	N	Window Speed Field

The **Socket** field identifies the logical socket containing the PC Card which has an undefined or incorrectly defined region.

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

Bit 0	Memory type (set = attribute)
Bit 1	Delete Region (set = true)
Bits 2 -- 7	RESERVED (Reset to zero)
Bit 8	Virtual Region (set = true)
Bits 12 -- 15	RESERVED (Reset to zero)

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

Delete Region is set to one when Card Services should remove the last region for this PC Card's memory type from its internal region table. If the region specfied is not the last region for this PC Card's memory type, BAD_OFFSET is returned. If an MTD is currently registered for this region, BAD_OFFSET is returned.

> **Note:**
> **If Delete Region is set to one, all fields after Region Size (offset 8) are ignored.**

The **Card Offset** through **JEDEC ID** fields are the same as defined in **GetFirstPartition**.

Once all of the regions on a PC Card requiring modification are successfully updated, the client should issue a **ResetCard** request to notify registered MTDs they should re-evaluate whether they wish to handle any region on the card.

The **BiasOffset** field is used by Card Services to compute the address to pass to the supporting MTD. When a read, write, copy or erase request is made by a client, Card Services subtracts this value from the (relative) value passed by the client and adds the base offset passed to **OpenMemory** before giving the request to the MTD. This field should normally be zero for regions that are not virtual.

The value in the **Card Offset** field is used by Card Services during **OpenMemory** requests to determine the MTD supporting access to the memory.

A new region may be added by using a region number equal to the current number of regions on the PC Card.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to twenty-five (25)
BAD_ATTRIBUTE	Virtual region cannot be set due to existing physical region
BAD_OFFSET	Region offset is invalid
BAD_SIZE	Region size is invalid
BAD_SOCKET	Socket is invalid
BAD_SPEED	Speed is not supported
NO_CARD	No PC Card in socket

5.52 ValidateCIS (2BH)

```
CardServices(ValidateCIS, null, null, ArgLength, ArgPointer)
```

This function validates the Card Information Structure on the PC Card in the specified socket.

Offset	Field	Size	Type	Value	Detail/Description
0	Socket	2	I	N	Logical Socket
2	Chains	2	O	N	Number of chains validated

The Socket field identifies the logical socket that contains the PC Card.

The Chains field returns the number of valid tuple chains located in the CIS. If zero (0) is returned, the CIS is not valid.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to four (4)
BAD_SOCKET	Socket is invalid
NO_CARD	No PC Card in socket

5.53 VendorSpecific (34H)

`CardServices(VendorSpecific, null, null, ArgLength, ArgPointer)`

This function is used to make vendor specific Card Services requests.

Offset	Field	Size	Type	Value	Detail/Description
0	InfoLen	2	I/O	N	Length of returned information in packet.
2	VendorData	N	I/O	N	Vendor Specific Data

The **InfoLen** specifies the length of the valid portion of the VendorData passed and returned.

The **VendorData** is a vendor specific structure.

Return Codes:

BAD_ARG_LENGTH	ArgLength is less than two (2)
«Vendor specific»	«Vendor specifies valid return codes»

5.54 WriteMemory (24H)

```
CardServices(WriteMemory, MemoryHandle, buffer, ArgLength, ArgPointer)
```

This function writes data to a PC Card via the specified MemoryHandle. The MemoryHandle returned by OpenMemory is passed in the Handle argument. The pointer to the data buffer that contains the data to be written to the PC Card is passed in the Pointer argument.

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

The **Card Offset** is a relative offset from the physical offset provided to the **OpenMemory** request used to obtain the **MemoryHandle**. It is the location on the PC Card where the data should be written.

The **Count** field is the number of bytes to write to the PC Card.

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

Bits 0 -- 1	RESERVED (reset to zero)
Bit 2	Disable Erase (set to one = true)
Bit 3	Verify
Bits 4 -- 15	RESERVED (reset to zero)

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

If the **MemoryHandle** identifies attribute memory, the client must access the attribute memory in an appropriate way. No special processing is done, i.e. all bytes requested are transferred, not just even bytes. This will typically mean that only single bytes on even address can be successfully written.

When used in a processor mode that supports segmentation (e.g. Intel 80286 protected mode operation), all bytes transferred are required to be contained within the segment referenced by the pointer argument. This limits the count requested to be less than or equal to the maximum segment size.

Return Codes:

BAD_ARG_LENGTH	ArgLength is not equal to ten (10)
BAD_HANDLE	Invalid memory area handle
BAD_OFFSET	Invalid destination offset
BAD_SIZE	Size of area to write is invalid
NO_CARD	No PC Card in socket
WRITE_FAILURE	Unable to write media
WRITE_PROTECTED	Media is write-protected

APPENDIX - A

FUNCTION CODES

A.1 Function Codes

Function	Code
Client Services Functions	
GetCardServicesInfo	0B$_H$
RegisterClient	10$_H$
DeregisterClient	02$_H$
GetStatus	0C$_H$
ResetCard	11$_H$
SetEventMask	31$_H$
GetEventMask	2E$_H$
Resource Management Functions	
RequestIO	1F$_H$
ReleaseIO	1B$_H$
RequestIRQ	20$_H$
ReleaseIRQ	1C$_H$
RequestWindow	21$_H$
ReleaseWindow	1D$_H$
ModifyWindow	17$_H$
MapMemPage	14$_H$
RequestSocketMask	22$_H$
ReleaseSocketMask	2F$_H$
RequestConfiguration	30$_H$
GetConfigurationInfo	04$_H$
ModifyConfiguration	27$_H$
ReleaseConfiguration	1E$_H$
Bulk Memory Services Functions	
OpenMemory	18$_H$
ReadMemory	19$_H$
WriteMemory	24$_H$
CopyMemory	01$_H$
RegisterEraseQueue	0F$_H$
CheckEraseQueue	26$_H$
DeregisterEraseQueue	25$_H$
CloseMemory	00$_H$
Client Utilities Functions	
GetFirstTuple	07$_H$
GetNextTuple	0A$_H$
GetTupleData	0D$_H$
GetFirstRegion	06$_H$
GetNextRegion	09$_H$
GetFirstPartition	05$_H$
GetNextPartition	08$_H$
Advanced Client Services Functions	
ReturnSSEntry	23$_H$
MapLogSocket	12$_H$
MapPhySocket	15$_H$
MapLogWindow	13$_H$
MapPhyWindow	16$_H$
RegisterMTD	1A$_H$
RegisterTimer	28$_H$
SetRegion	29$_H$
ValidateCIS	2B$_H$
RequestExclusive	2C$_H$
ReleaseExclusive	2D$_H$
GetFirstClient	0E$_H$
GetNextClient	2A$_H$
GetClientInfo	03$_H$
AddSocketServices	32$_H$
ReplaceSocketServices	33$_H$
VendorSpecific	34$_H$
AdjustResourceInfo	35$_H$

APPENDIX - B

EVENT CODES

B.1 Event Codes

Event	Code	Source	Client(s)	Registered By
PM_RESUME	0BH	Card Services	Socket	RequestSocket
PM_SUSPEND	0CH	Card Services	Socket	RequestSocket
BATTERY_DEAD	01H	Hardware	Socket	RequestSocket
BATTERY_LOW	02H	Hardware	Socket	RequestSocket
CARD_INSERTION [A]	40H	DeregisterMTD	MTDs	RegisterClient
CARD_INSERTION	40H	Hardware	All	RegisterClient
CARD_INSERTION [A]	40H	RequestExclusive	Requester	RequestExclusive
CARD_INSERTION [A]	40H	RegisterClient	Requester	RegisterClient
CARD_INSERTION [A]	40H	RequestExclusive	All	RegisterClient
CARD_LOCK	03H	Hardware	Socket	RequestSocket
CARD_READY	04H	Hardware	Socket	RequestSocket
CARD_REMOVAL	05H	Hardware	Socket	RequestSocket
CARD_REMOVAL [A]	05H	ReleaseExclusive	Socket	RequestSocket
CARD_RESET	11H	ResetCard	Socket	RequestSocket
CARD_UNLOCK	06H	Hardware	Socket	RequestSocket
EJECTION_COMPLETE	07H	Hardware	Socket	RequestSocket
EJECTION_REQUEST	08H	Hardware	Socket	RequestSocket
ERASE_COMPLETE	81H	Queued Erase	Requestor	RegisterEraseQueue
EXCLUSIVE_COMPLETE	0DH	RequestExclusive	Requester	RequestExclusive
EXCLUSIVE_REQUEST	0EH	RequestExclusive	Socket	RequestSocket
INSERTION_COMPLETE	09H	Hardware	Socket	RequestSocket
INSERTION_REQUEST	0AH	Hardware	Socket	RequestSocket
REGISTRATION_COMPLETE	82H	RegisterClient	Requester	RegisterClient
RESET_COMPLETE	80H	ResetCard	Requester	ResetCard
RESET_PHYSICAL	0FH	ResetCard	Socket	RegisterClient
RESET_REQUEST	10H	ResetCard	Socket	RegisterClient
MTD_REQUEST	12H	Card Services	MTD	RegisterClient
CLIENT_INFO	14H	GetClientInfo	Provider	RegisterClient
TIMER_EXPIRED	15H	Hardware	Requester	RegisterTimer
SS_UPDATED	16H	Card Services	All	RegisterClient
WRITE_PROTECT	*17H*	*Hardware*	*All*	*RegisterClient*

See the description of the table columns on the next page.

APPENDIX - C

RETURN CODES

C.1 Return Codes

Return Code	Value	Description
SUCCESS	00H	The request succeeded.
BAD_ADAPTER	01H	Specified adapter is invalid
BAD_ATTRIBUTE	02H	Value specified for attributes field is invalid
BAD_BASE	03H	Specified base system memory address is invalid
BAD_EDC	04H	Specified EDC generator is invalid
«Reserved»	*05H*	*«Reserved for historical purposes»*
BAD_IRQ	06H	Specified IRQ level is invalid
BAD_OFFSET	07H	Specified PC Card memory array offset is invalid
BAD_PAGE	08H	Specified page is invalid
READ_FAILURE	09H	Unable to complete read request
BAD_SIZE	0AH	Specified size is invalid
BAD_SOCKET	0BH	Specified socket is invalid (logical or physical)
«Reserved»	*0CH*	*«Reserved for historical purposes»*
BAD_TYPE	0DH	Window or interface type specified is invalid
BAD_VCC	0EH	Specified Vcc power level index is invalid
BAD_VPP	0FH	Specified Vpp1 or Vpp2 power level index is invalid
«Reserved»	*10H*	*«Reserved for historical purposes»*
BAD_WINDOW	11H	Specified window is invalid
WRITE_FAILURE	12H	Unable to complete write request
«Reserved»	*13H*	*«Reserved for historical purposes»*
NO_CARD	14H	No PC Card in socket
UNSUPPORTED_FUNCTION	15H	Implementation does not support function
UNSUPPORTED_MODE	16H	Processor mode is not supported
BAD_SPEED	17H	Specified speed is unavailable
BUSY	18H	Unable to process request at this time - retry later
GENERAL_FAILURE	19H	An undefined error has occurred
WRITE_PROTECTED	1AH	Media is write-protected
BAD_ARG_LENGTH	1BH	ArgLength argument is invalid
BAD_ARGS	1CH	Values in Argument Packet are invalid
CONFIGURATION_LOCKED	1DH	A configuration has already been locked
IN_USE	1EH	Requested resource is being used by a client
NO_MORE_ITEMS	1FH	There are no more of the requested item
OUT_OF_RESOURCE	20H	Card Services has exhausted resource
BAD_HANDLE	21H	ClientHandle is invalid

Note:
Return Codes common to Socket Services use the same values. Italicized entries are reserved for historical purposes and should not be used. Return codes above 19H are unique to Card Services.

APPENDIX - D

PROCESSOR BINDINGS

PROCESSOR BINDINGS

Five general arguments are defined for making Card Services requests:

- **Function**,
- **Handle**,
- **Pointer**,
- **ArgLength**, and
- **ArgPointer**.

A **Status** argument returns information from a Card Services request.

For a given processor, specific registers are defined for these general Card Services arguments.

The method of requesting a Card Services function is also specific to a given processor and operating system environment and is documented under each binding.

D.1 Add/ReplaceSocketServices

The definition of the Attributes field of the **Add/ReplaceSocketServices** requests provides support for Socket Services implementations for Intel 8086 compatible processors. The values defined for the **Attributes** field is:

0	Real Mode
1	16:16 Protect Mode
2	16:32 Protect Mode
3	0:32 Protect Mode (Flat Model)
-	Other values are reserved.

Other values will be defined for other processors as required.

D.2 Intel 8086 Real Mode DOS

Card Services argument binding:

[AL]	= Function
[DX]	= Handle
[DI]:[SI]	= Pointer, [DI] is 16 bit segment, [SI] is 16 bit offset
[CX]	= ArgLength
[ES]:[BX]	= ArgPointer, [ES] is 16 bit segment, [BX] is 16 bit offset
[AX]	= Status
[CF]	= reset to zero when request was successful, set to one when request failed

All entry registers are preserved unless the same register is used for a return value.

All calls and returns are FAR.

All pointers are SEGMENT16:OFFSET16.

Card Services is invoked by INT 1AH with [AH] = AFH.

The **RegisterClient** argument packet ClientData field is defined to be:

Offset	Description
0	16 bit data determined by client
2	16 bit segment for client data area
4	16 bit offset determined by client
6	16 bit Reserved (Reset to zero)

The SEGMENT:OFFSET pointer in Bytes 2-5 will typically be set by the client to the client internal data area. Since Card Services passes these values to a client callback, this gives the client addressability to its internal data area on entry to the callback handler.

Callback handler arguments are:

[AL]	= Function
[CX]	= Socket
[DX]	= Info
[DI]	= first word of **RegisterClient** argument packet ClientData field
[DS]	= second word of **RegisterClient** argument packet ClientData field
[SI]	= third word of **RegisterClient** argument packet ClientData field
[SS]:[BP]	= MTDRequest
[ES]:[BX]	= Buffer
[BX]	= Misc (when no buffer is specified)
[AX]	= Status on return to Card Services
[CF]	= returned reset to zero when the callback completed successfully, set to one if Status is not SUCCESS.

The address returned by ReturnSSEntry() must be invoked using a CALL FAR instruction. The client should not assume this entry point is the same as that returned by the Socket Services GetSetSSAddr() function. It may be the address of a routine located within Card Services.

D.3 Intel 80286 Protect Mode (SAMPLE)

Argument binding:

[AL]	= Function
[DX]	= Handle
[DI]:[SI]	= Pointer, [DI] is 16 bit selector, [SI] is 16 bit offset
[CX]	= ArgLength
[ES]:[BX]	= ArgPointer, [ES] is 16 bit selector, [BX] is 16 bit offset
[AX]	= Status
[CF]	= reset to zero when request was successful, set to one when request failed

All calls and returns are FAR.

The invocation of Card Services in this mode is not defined in this release of Card Services.

The callback binding is not defined in this release of Card Services.

The **RegisterClient** argument packet ClientData field is defined to be:

Offset	Description
0	16 bit data determined by client
2	16 bit selector for client data area
4	16 bit offset determined by client
6	16 bit Reserved (Reset to zero)

The address returned by ReturnSSEntry() must be invoked using a CALL FAR instruction. The client should not assume this entry point is the same as that returned by the Socket Services GetSetSSAddr() function. It may be the address of a routine located within Card Services.

D.4 Intel 80386 Protect Mode (SAMPLE)

Argument binding:

`[AL]`	= Function
`[EDX]`	= Handle
`[EDI]:[ESI]`	= Pointer, `[EDI]` is 16-bit selector, `[ESI]` is 32 bit offset
`[ECX]`	= ArgLength
`[ES]:[EBX]`	= ArgPointer, `[ES]` is 16-bit selector, `[EBX]` is 32 bit offset
`[EAX]`	= Status
`[CF]`	= reset to zero when request was successful, set to one when request failed

All calls and returns are FAR.

The invocation of Card Services in this mode is not defined in this release of Card Services.

The callback binding is not defined in this release of Card Services.

The **RegisterClient** argument packet ClientData field is defined to be:

Offset	Description
0	16-bit data determined by client
2	16-bit selector for client data area
4	32-bit offset determined by client

The address returned by ReturnSSEntry() must be invoked using a CALL FAR instruction. The client should not assume this entry point is the same as that returned by the Socket Services GetSetSSAddr() function. It may be the address of a routine located within Card Services.

D.5 Intel 80386 Flat Protect Mode (SAMPLE)

Argument binding:

[AL]	= Function
[EDX]	= Handle
[ESI]	= Pointer, [ESI] is 32-bit offset
[ECX]	= ArgLength
[EBX]	= ArgPointer, [EBX] is 32-bit offset
[EAX]	= Status
[CF]	= reset to zero when request was successful, set to one when request failed

All calls and returns are NEAR.

The invocation of Card Services in this mode is not defined in this release of Card Services.

The callback binding is not defined in this release of Card Services.

The RegisterClient argument packet ClientData field is defined to be:

Offset	Description
0	16 bit data determined by client
2	16 bit Reserved (Reset to zero)
4	32 bit offset determined by client

The address returned by ReturnSSEntry() must be invoked using a CALL NEAR instruction. The client should not assume this entry point is the same as that returned by the Socket Services GetSetSSAddr() function. It may be the address of a routine located within Card Services.

D.6 Intel 80286 Protect Mode OS/2

Card Services argument binding:

[AL]	*= Function*
[DX]	*= Handle*
[DI]:[SI]	*= Pointer, [DI] is 16 bit selector, [SI] is 16 bit offset*
[CX]	*= ArgLength*
[ES]:[BX]	*= ArgPointer, [ES] is 16 bit selector, [BX] is 16 bit offset*
[AX]	*= Status*
[CF]	*= reset to zero when request was successful, set to one when request failed*

All entry registers are preserved unless the same register is used for a return value.

All calls and returns are FAR.

All pointers are SELECTOR16:OFFSET16.

Card Services is invoked via an Inter Device Communications (IDC) interface. A client device driver obtains the IDC entry point address by invoking the AttachDD DevHlp function with the device driver name "PCMCIA$ " (PCMCIA$ followed by a blank space).

*The **RegisterClient** argument packet ClientData field is defined to be:*

Offset	Description
0	16 bit data determined by client
2	16 bit selector for client data area
4	16 bit offset determined by client
6	16 bit Reserved (Reset to zero)

The SELECTOR:OFFSET pointer in Bytes 2-5 will typically be set by the client to the client internal data area. Since Card Services passes these values to a client callback, this gives the client addressability to its internal data area on entry to the callback handler.

Callback handler arguments are:

[AL]	*= Function*
[CX]	*= Socket*
[DX]	*= Info*
[DI]	*= first word of **RegisterClient** argument packet ClientData field*
[DS]	*= second word of **RegisterClient** argument packet ClientData field*
[SI]	*= third word of **RegisterClient** argument packet ClientData field*
[SS]:[BP]	*= MTDRequest*
[ES]:[BX]	*= Buffer*
[BX]	*= Misc (when no buffer is specified)*
[AX]	*= Status on return to Card Services*
[CF]	*= returned reset to zero when the callback completed successfully, set to one if Status is not SUCCESS.*

The address returned by ReturnSSEntry() must be invoked using a CALL FAR instruction. The client should not assume this entry point is the same as that returned by the Socket Services GetSetSSAddr() function. It may be the address of a routine located within Card Services.

APPENDIX - E

MTD HELPER FUNCTION REFERENCE

MTD HELPER FUNCTION REFERENCE

These functions are defined for Intel 8086 compatible DOS and Intel 80286 compatible OS/2 environments. Address include SEGMENT16 values for Intel 8086 DOS environments and SELECTOR16 values for Intel 80286 OS/2 environments. ~~These functions are only defined for an Intel 8086 compatible DOS environment. Other processor and OS bindings will be added as required~~.

These functions are requested via the pointer in the Media Access Table passed to MTDs. These functions should only be used by an MTD during its processing of a read, write, or erase request.

All MTD Helper functions pass information in the following registers:

[AX] = MTD Helper Entry Value

All MTD Helper functions return [CF] reset to zero if the request was successfully processed. [CF] is set to one if the request was not completed for some reason. [AH] is returned with a value that indicates more detailed information of the request success/failure. This release of the Card Services Interface Specification does not define the values of the return codes.

E.1 MTDModifyWindow (00H)

This function changes the mapping for the window descriptor to the currently specified values. The values in the request packet for socket, window, system base address, and window size must not be changed from the values originally passed to the MTD by **MTDRequestWindow**.

Entry:

[AX]	= 00H — MTDModifyWindow	
[DX]	= WindowHandle (returned by RequestWindow)	
[BH]	= Attributes	
	Bit 0	Reserved by Card Services (ignored by MTD)
	Bit 1	Type of Memory (set = attribute, reset = common)
	Bit 2 ·· 7	Reserved (reset to zero)
[BL]	= Access Speed	
	Bit 0 ·· 2	Device Speed Code, if Speed Mantissa is zero
	Bit 0 ·· 2	Speed Exponent, if Speed Mantissa is non-zero
	Bit 3 ·· 6	Speed Mantissa
	Bit 7	Use WAIT, if available
[DI]:[SI]	= Card Offset	

Exit:

Returns [CF] reset to zero if function completed successfully. [CF] is set to one if the changes to the window mapping could not be made.

The window must be released before the MTD returns from Card Services.

E.2 MTDReleaseWindow (01H)

This function returns to Card Services a window descriptor that an MTD had requested.

Entry:

[AX]	= 01H — MTDReleaseWindow
[DX]	= WindowHandle (returned by **RequestWindow**)

Exit:

Returns [CF] reset to zero if function completed successfully. [CF] is set to one if this window had not been previously requested.

The window descriptor is the same as for **MTDModifyWindow**.

E.3 MTDRequestWindow (02H)

This function returns a window descriptor that an MTD can use to control direct access to memory.

Entry:

[AX]	= 02H — MTDRequestWindow	
[BH]	= Attributes	
	Bit 0	Reserved by Card Services (ignored by MTD)
	Bit 1	Type of Memory (set = attribute, reset = common)
	Bit 2	Card offset alignment on size boundary (output only)
	Bit 3 -- 7	Reserved (reset to zero)
[BL]	= Access Speed	
	Bit 0 -- 2	Device Speed Code, if Speed Mantissa is zero
	Bit 0 -- 2	Speed Exponent, if Speed Mantissa is non-zero
	Bit 3 -- 6	Speed Mantissa
	Bit 7	WAIT required
[CX]	= Window size (in 4KB units), if zero, largest size available will be returned	
[DX]	= Logical socket	

Exit:

[DX]	= WindowHandle (returned by **RequestWindow**)
[ES]:[DI]	= System address for Window
[CX]	= Window size
[BH]	= Bit 2 indicates the PC Card offset alignment requirements for PC Card mapping. If reset to zero, the PC Card offset can be any 4KB boundary. If set to one, the PC Card offset must be a window size multiple.

Returns [CF] reset to zero if function completed successfully. [CF] is set to one if no windows are available.

The window descriptor is the same as for **MTDModifyWindow**.

E.4 MTDSetVpp (03H)

This function sets the Vpp levels for the socket to the requested values.

Entry:

[AX]	= 03H — MTDSetVpp
[BL]	= Vpp1 Voltage to set in 0.1V increments (0.0v - 25.5v)
[BH]	= Vpp2 Voltage to set in 0.1V increments (0.0v - 25.5v)
[DX]	= Logical Socket

Exit:

An MTD must set Vpp1 and Vpp2 back to Vcc after completing its request. Returns [CF] reset to zero if the function completed successfully and the voltage is stable. [CF] is set to one if the voltage could not be set.

E.5 MTDRDYMask (04H)

This function enables and disables the RDY event mask to allow an MTD to avoid generating extraneous RDY events in the system.

Entry:

[AX]	= 04H — MTDRDYMask
[BL]	= set to one to enable, reset to zero to disable

Exit:

No exit parameters.

APPENDIX - F

MEDIA ACCESS FUNCTION~~S FOR INTEL 8086~~ REFERENCE

MEDIA ACCESS FUNCTION~~S FOR INTEL 8086~~ REFERENCE

F.1 CardSetAddress

These functions are defined for Intel 8086 compatible DOS and Intel 80286 compatible OS/2 environments. Address include SEGMENT16 values for Intel 8086 DOS environments and SELECTOR16 values for Intel 80286 OS/2 environments.

Entry:

[AX]:[DX] = PC Card absolute address

[BH] = bit mapped attributes

Bit 0 = attribute memory (set to 1)
common memory (reset to 0)

Bit 1 = read requests (reset to 0)
write requests (set to 1)

Bit 2 -- 7 = reserved (reset to 0)

[BL] = access speed

[CX] = logical socket

Exit:

[DX] = MAT transfer token value for use by read/write requests

[DS]:[SI] = MAT transfer token value for use by read requests

— or —

[ES]:[DI] = MAT transfer token value for use by write requests

[CX] = maximum number of bytes that can be transferred when using the autoincrement addressing before another CardSetAddress is required

The processor direction flag is cleared which causes string instructions to increment system addresses.

Autoincrement PC Card memory addressing is turned on.

CardSetAddress must be called before performing any PC Card memory access with the other media access requests. CardSetAddress controls the adapter to allow access to PC Card memory via the other media access requests.

F.2 CardSetAutoInc

Entry:

[AX] = reset to 0 turns off autoincrement, set to 1 turns on autoincrement

Exit:

The processor direction flag is cleared which causes string instructions to increment system addresses.

CardSetAutoInc controls the adapter to turn on or off autoincrementing. If autoincrementing is turned on, the Card requests with AI at the end requests must be used (see below). If autoincrementing is turned off, the non Card requests without the AI at the end must be used.

F.3 CardReadByte

F.4 CardReadWord

F.5 CardReadByteAI

F.6 CardReadWordAI

Entry:

[DX]	= MAT transfer token value
[DS]:[SI]	= MAT transfer token value

Exit:

[AX]	= data read for ReadWord
	— or —
[AL]	= data read for ReadByte
[DX]	= new MAT transfer token value
[DS]:[SI]	= new MAT transfer token value

These routines read a byte or word from PC Card memory. The CardRead routines with AI at the end must be used if autoincrementing is turned on, and the CardRead routines without AI at the end must be used if autoincrementing is turned off.

F.7 CardReadWords

F.8 CardReadWordsAI

Entry:

[CX]	= number of words to transfer
[DX]	= MAT transfer token value
[DS]:[SI]	= MAT transfer token value
[ES]:[DI]	= pointer to buffer for data read

Exit:

[ES]	= unchanged
[DI]	= input [DI] + input [CX]
[CX]	= number of bytes remaining
[DX]	= new MAT transfer token value
[DS]:[SI]	= new MAT transfer token value

CardReadWords and **CardReadWordsAI** read words from PC Card memory into the supplied buffer. The current PC Card memory address must be word aligned. **CardReadWordsAI** can only be used if autoincrementing is turned on.

F.9 CardWriteByte

F.10 CardWriteWord

F.11 CardWriteByteAI

F.12 CardWriteWordAI

Entry:

[AX]	= data to write for WriteWord
	— or —
[AL]	= data to write for WriteByte
[DX]	= MAT transfer token value
[ES]:[DI]	= MAT transfer token value

Exit:

[DX]	= new MAT transfer token value
[ES]:[DI]	= new MAT transfer token value

These routines write a byte or word to PC Card memory. The CardWrite routines followed by AI must be used if autoincrementing is turned on, and the CardWrite without AI at the end must be used if autoincrementing is turned off.

F.13 CardWriteWords

F.14 CardWriteWordsAI

Entry:

[CX]	= number of words to write
[DX]	= MAT transfer token value
[ES]:[DI]	= MAT transfer token value
[DS]:[SI]	= pointer to buffer for data to write

Exit:

[DS]	= unchanged
[SI]	= input [SI] + input [CX]
[CX]	= number of bytes remaining
[DX]	= new MAT transfer token value
[ES]:[DI]	= new MAT transfer token value

CardWriteWords and CardWriteWordsAI writes words to PC Card memory. The current PC Card memory address must be word aligned. CardWriteWordsAI can only be used if autoincrementing is turned on.

APPENDIX - G

ARGUMENT USAGE REFERENCE

ARGUMENT USAGE REFERENCE

This table indicates which arguments are used for each Card Services request. The Handle argument indicates input value followed by output value (input/output). If both values are the same, one value is listed. A ✔ indicates that the argument is used for the listed request. No entry for an argument indicates that the request does not use that argument. The value of the Status argument is returned by Card Services to the requesting client.

Request	Function	Handle	Pointer	ArgLen	ArgPtr	Status
AddSocketServices	✔		entry	✔	✔	✔
AdjustResourceInfo	✔			✔	✔	✔
CheckEraseQueue	✔	Queue		✔		✔
CloseMemory	✔	Memory/–				✔
CopyMemory	✔	Memory		✔	✔	✔
DeregisterClient	✔	Client/–				✔
DeregisterEraseQueue	✔	Queue/–				✔
GetCardServicesInfo	✔			✔	✔	✔
GetClientInfo	✔	Client		✔	✔	✔
GetConfigurationInfo	✔	–/Client		✔	✔	✔
GetEventMask	✔	Client		✔	✔	✔
GetFirstClient	✔	–/Client		✔	✔	✔
GetFirstPartition	✔			✔	✔	✔
GetFirstRegion	✔	–/Client		✔	✔	✔
GetFirstTuple	✔			✔	✔	✔
GetNextClient	✔	Client/Client		✔	✔	✔
GetNextPartition	✔			✔	✔	✔
GetNextRegion	✔	–/Client		✔	✔	✔
GetNextTuple	✔			✔	✔	✔
GetStatus	✔			✔	✔	✔
GetTupleData	✔			✔	✔	✔
MapLogSocket	✔			✔	✔	✔
MapLogWindow	✔	Window		✔	✔	✔
MapMemPage	✔	Window		✔	✔	✔
MapPhySocket	✔			✔	✔	✔
MapPhyWindow	✔	–/Window		✔	✔	✔
ModifyConfiguration	✔	Client		✔	✔	✔
ModifyWindow	✔	Window		✔	✔	✔
OpenMemory	✔	Client/Memory		✔	✔	✔
ReadMemory	✔	Memory	buffer	✔	✔	✔

Request	Function	Handle	Pointer	ArgLen	ArgPtr	Status
RegisterClient	✔	–/Client	entry	✔	✔	✔
RegisterEraseQueue	✔	Client/Queue	queuehead			✔
RegisterMTD	✔	Client		✔	✔	✔
RegisterTimer	✔	Client/Timer		✔	✔	✔
ReleaseConfiguration	✔	Client		✔	✔	✔
ReleaseExclusive	✔	Client		✔	✔	✔
ReleaseIO	✔	Client		✔	✔	✔
ReleaseIRQ	✔	Client		✔	✔	✔
ReleaseSocketMask	✔	Client		✔	✔	✔
ReleaseWindow	✔	Window/–				✔
ReplaceSocketServices	✔		entry	✔	✔	✔
RequestConfiguration	✔	Client		✔	✔	✔
RequestExclusive	✔	Client		✔	✔	✔
RequestIO	✔	Client		✔	✔	✔
RequestIRQ	✔	Client		✔	✔	✔
RequestSocketMask	✔	Client		✔	✔	✔
RequestWindow	✔	Client/Window		✔	✔	✔
ResetCard	✔	Client		✔	✔	✔
ReturnSSEntry	✔		–/entry	✔	✔	✔
SetEventMask	✔	Client		✔	✔	✔
SetRegion	✔			✔	✔	✔
WriteMemory	✔	Memory	buffer	✔	✔	✔
ValidateCIS	✔			✔	✔	✔

APPENDIX - H

CLIENT CALLBACK ARGUMENT USAGE

CLIENT CALLBACK ARGUMENT USAGE

This table indicates which arguments are used for each Card Services callback procedure. The **Misc** argument indicates input value followed by output value (input/output). If both values are the same, one value is listed. A ✔ indicates that the argument is used for the listed request. No entry for an argument indicates that the request does not use that argument. The value of the **Status** argument is returned by a client to Card Services.

Callback	Function	Socket	Info	Client Data	MTD Request	Buffer	Misc	Status
Insertion	✔	✔		✔				✔
RegistrationComplete	✔			✔				✔
StatusChange	✔	✔		✔				✔
Ejection/Insertion	✔	✔		✔				✔
Exclusive	✔	✔	status	✔				✔
Reset	✔	✔	status	✔				✔
ClientInformation	✔			✔		✔		✔
EraseCompletion	✔	✔	entry num	✔			Queue Handle	✔
MTDRequest	✔	✔		✔	✔	✔		✔
Timer	✔			✔			Timer Handle	✔
NewSocketServices	✔	✔	SSinfo	✔				✔

Callback	Event
Insertion	CARD_INSERTION
RegistrationComplete	REGISTRATION_COMPLETE
Status Change	BATTERY_LOW CARD_LOCK CARD_READY BATTERY_DEAD CARD_UNLOCK CARD_REMOVAL
Ejection/Insertion	EJECTION_REQUEST EJECTION_COMPLETE INSERTION_REQUEST INSERTION_COMPLETE
Exclusive	EXCLUSIVE_REQUEST RESET_PHYSICAL CARD_RESET RESET_COMPLETE
ClientInformsation	CLIENT_INFO
EraseCompletion	ERASE_COMPLETE
MTDRequest	MTD_REQUEST
Timer	TIMER_EXPIRED
NewSocketServices	SS_UPDATED

APPENDIX - I

OS CRITICAL SECTION HANDLING

OS CRITICAL SECTION HANDLING

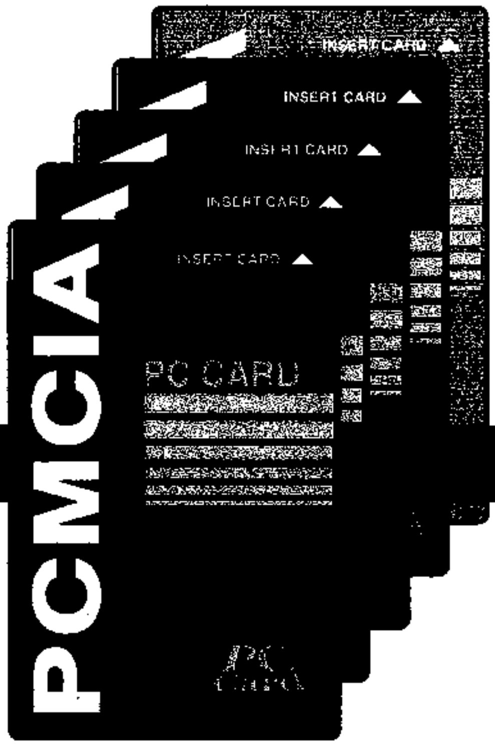
Card Services is designed with the intention that a client should never find a situation in which a Card Services call is failed because Card Services is not enterable. However, in a DOS system a TSR or application which performs Card Services calls can be in the middle of a Card Services call and the system can switch to another task because it is running under Windows, Task Swapper, DesqView or a similar environment. The new task can then attempt to use the file system client which in turn uses Card Services. This causes a critical error (Abort, Retry, Ignore, Fail) or other catastrophic error to be returned because the file system will find Card Services unenterable, since the previous task is still in Card Services.

For clients such as file systems (block/character device drivers, etc.), this is not a problem because access to Card Services is synchronized through MS-DOS's own critical section handling, and the task-switching environments have all been designed to not switch away from a task during the middle of an MS-DOS function call. However, TSR's and generalized applications do not have the same synchronization, and it is therefore necessary for any TSR or application which makes Card Services calls to encapsulate *ALL* such calls within the Enter/Leave Critical Section APIs appropriate to the detected task switching environment.

Clients in task-switching environments should use the Critical Section handler APIs appropriate to such environments around *EVERY* call to Card Services. Information about these APIs can be found in following:

Windows:	The Windows Device Development Kit (DDK), Virtual Device Adaptation Guide, Appendix D — Windows INT 2F API.
MS-DOS Task: Switcher	The MS-DOS 5.0 Programmer's Reference, Section 7.10, Task Swapper Reference, and INT 2F Function 4Bxx.
DesqView:	The DesqView User's Manual, Appendix J contains assembly code fragments to allow the creation of DesqView aware applications, including the necessary API calls.

PERSONAL
COMPUTER
MEMORY CARD
INTERNATIONAL
ASSOCIATION



PCMCIA STANDARDS

SECTION - 1

INTRODUCTION

INTRODUCTION

1.1 Purpose

This specification defines the standard method for incorporating an ATA mass storage protocol peripheral on a PCMCIA PC Card. This specification supplements the definitions of an ATA mass storage peripheral found in the ATA document and the definition of a PCMCIA PC Card found in the PC Card document.

The PC Card ATA protocol described in this document is compatible with existing PCMCIA defined socket hardware without any changes or additional pins. PC Card ATA mass storage cards shall be implemented to operate as I/O devices in conformance with Release 2.01 of the PC Card document. The cards are also permitted to provide a memory mapped configuration compatible with socket hardware defined in Release 1.0 of the PC Card document.

This document describes the electrical and software interfaces for an ATA PC Card. The standard address mappings for an ATA PC Card are also described.

1.2 Scope

This document is intended to be used together with the PC Card document and the ATA document. It is intended to highlight those areas of implementation in which the PC Card document and the ATA document conflict. In addition, an indication is made of areas within the ATA document which are modified for operation in a PC Card environment. Both mandatory and optional specifications are presented.

In the event of a conflict between one of the base documents (PC Card document or ATA document) and this document, the interpretation of this document shall prevail if and only if this document specifies that a conflict exists between the documents.

1.3 Related Documents

There are two documents upon which this document is based and which are required for understanding and implementing a PC Card ATA mass storage peripheral. The base documents are as follows:

a) PCMCIA, *PC Card Standard*, Release 2.01, November 1992, Personal Computer Memory Card International Association. This is referred to as the PCMCIA PC Card document.

b) CBEMA, *ATA (AT Attachment) ANSI Draft Standard*, Revision 3.1, August 15, 1992, Document Number X3T9.2/90-143, Computer and Business Equipment Manufacturer's Association. This is referred to as the ATA document.

Several additional documents may assist the reader in developing the hardware and software to utilize an ATA PC Card within a compatible system. These supplementary documents are as follows:

a) PCMCIA, *Socket Services Interface Specification*, Release 2.0, November, 1992, Personal Computer Memory Card International Association.

b) PCMCIA, *Card Services Interface Specification*, Release 2.0, November, 1992, Personal Computer Memory Card International Association.

1.4 Conventions

This section is intended to give general descriptions of notational conventions used in this document.

A glossary is provided in Appendix A of this document with an extensive set of definitions. In many cases, more detailed information about these terms may be found in the PCMCIA PC Card document or the ATA document. These two documents should be consulted for more detailed and precise definitions of terms.

1.4.1 Signal Naming

Signals which are asserted when a logic low is present have a "-" prefix to the signal name. Signals which are asserted when a logic high is present have no "-" prefix. Signals which are not logic signals, such as VCC, are also shown with no "-" prefix.

All signals, except for Ready/-Busy, are named with respect to their asserted state as follows:

a) Each signal which is not a logic signal, such as Vcc, has a name which does not begin with a "-" character.

b) Each logic signal whose name does not begin with a "-" character has logic high as the asserted state and logic low as the negated state.

c) Each logic signal whose name begins with a "-" character has logic low as the asserted state and logic high as the negated state.

d) The Ready/-Busy signal shall be interpreted as being active in Busy state. Therefore, the signal is asserted by a a logic low and negated by a logic high.

1.4.2 Numeric Representation

Numbers are expressed as follows:

a) Individual bits are expressed as "0" for zero, "1" for one, or "X" for Don't Care.

b) Groups of bits (fields) are expressed in hexadecimal number which begin with a decimal digit and are followed by an "H." Each digit represents 4 bits and ranges from 0 to 9 and AH to FH for 0 to 15 (decimal) with an "X" being used for Don't Care. The number of bits in the field determines how many bits in the hexadecimal number are significant.

1.4.3 Bit Action Representation

Bits of a register are said to be Set when they are made equal to "1" and to be Cleared when they are made equal to "0."

SECTION - 2

OVERVIEW

OVERVIEW

This document details the requirements and considerations in implementing an ATA protocol mass storage peripheral within the PCMCIA environment.

The ATA protocol is followed except where the PCMCIA interface imposes conflicting constraints to traditional ATA Host Bus Adapters. This document describes how the ATA Protocol maps onto the PCMCIA interface. It resolves and clarifies the enhancements and restrictions which result from the use of the PCMCIA interface with the ATA Protocol.

Mandatory support is provided for the AT BIOS standard ATA I/O registers at 1F0H-1F7H, 3F6H-3F7H or 170H-177H, 376H-377H typically using IRQ14. These I/O port assignments are usable with pre-existing AT BIOS Device Drivers. Mandatory support is also provided for locating the I/O ports in a contiguous I/O window of at least 16 bytes which is decoded for the card by the socket. Cards can also be built that may be placed by the system in a 2 Kbyte host memory space by a dedicated driver.

The use of a well known, dominant, standard interface in mobile computers will guarantee system vendors an interface which can remain stable over the coming generations of silicon and rotating mass storage devices.

2.1 Feature Summary

a) The PC Card ATA protocol is based on the widely accepted and established ATA protocol which is a dominant standard for disk drives in mobile computers. It is also based on the PCMCIA interface standard which is the dominant emerging peripheral interface standard for mobile computing.

b) The ATA protocol is very familiar to both system designers and software developers.

c) This protocol allows ATA PC Cards to be "plug and play" in many existing systems and applications.

d) Compatible with existing ATA software.

e) The protocol fits easily into the architecture of desktop PC's as well as mobile computers.

2.2 Differences Between PC Card ATA and ATA.

a) The Diagnostic command runs only on the card which is addressed by the Drive-Head register when the Diagnostic Command is issued. This is because PCMCIA card interface does not provide for direct inter-drive communication (such as the ATA PDIAG and DASP signals). Therefore, unlike ATA, it is not possible when using the PCMCIA interface for Drive 0 to report status for both drives.

b) The PC Card ATA specification provides for two cards at a single address through the Twin Card option in CIS (PC Card document Section 5.2.8.3.10) and card enumeration using the Socket and Copy Register (PC Card document Section 4.15.4). The ATA specification provides card enumeration using a jumper or cable strap. The ATA signals PDIAG and DASP are not implemented in the PC Card ATA specification.

c) The PC Card ATA specification provides a Ready/-Busy signal which can be used to prevent the host from accessing the card's registers before the card is available following card detected power-on, hardware reset or PCMCIA soft reset. With an appropriate socket, this signal is also used while a card is configured in the memory mapped mode to provide a socket generated host interrupt on the Busy-to-Ready transition.

d) The PC Card ATA specification provides an ATA Soft Reset protocol described in section 6.1.

e) The implementation of the Index bit, IDX, in the Status Register and the Alternate Status Register is optional. If implemented, it shall be implemented as defined in the ATA document.

f) I/O ports 3F7H and 377H in the Primary and Secondary I/O mapped modes have a potential conflict with a floppy disk controller installed in the host. There is a potential problem with the protocol described in the ATA Document for sharing the Drive Address Register with a floppy disk controller when either the ATA peripheral or the floppy disk controller are accessed through the PCMCIA interface.

Refer to Appendix B for possible methods to avoid this problem.

g) The PCMCIA interface permits the host to access the ATA registers in more alternative ways than the traditional ATA host bus adapter allows. These alternatives arise from the presence of two card enable signals, -CE1 and -CE2, in addition to address line A0.

The PCMCIA allows accesses to registers at odd addresses with two different methods.

a) When address line A0 is 1 (logic high), if -CE1 is asserted and -CE2 is negated during the read or write cycle, then the byte of data at the odd address is transferred on signals D7 through D0 of the Data Bus.

b) Regardless of the state of address line A0 and of the state of -CE1, if -CE2 is asserted the byte of data at the odd address is transferred on signals D15 through D8 of the Data Bus. If -CE1 is also asserted, then a 16-bit word is accessed. If -CE1 is negated, then only the byte at the odd address is accessed.

A sixteen bit word of data is accessed when both -CE1 and -CE2 are active, regardless of the state of A0.

h) I/O accesses are constrained at the PCMCIA interface as follows:

a) The host shall perform all word (16-bit) I/O accesses with A0 = 0.

b) During a host's word access attempt, if a card returns -IOIS16 as active in response to the address on the bus then the host system is permitted to transfer 16-bits of data to the card in a single cycle, otherwise, the host system shall perform two 8-bit cycles: even byte then odd byte.

i) The ATA document specifies that the Data register is two bytes wide and is located at offset, or relative address, 0 while the Error and Feature registers are one byte wide and are located at offset 1 within the ATA registers. This results in an overlap of the address spaces between the Data register and Error-Feature register combination.

To permit hosts whose architectures do not permit word and byte registers to overlap to access all the registers of an ATA PC Card, the PC Card ATA specification provides a non-overlapping, duplicate copy of each of these registers in the Memory Mapped and Contiguous I/O mapped configurations. Within the 16 byte space occupied by the ATA registers in these configurations, the duplicate data register is located at offset 8H while the duplicate Error and Feature registers are located at offset 0DH.

Refer to section 4.2.13 for more information about the duplicate copies of these registers.

SECTION - 3

ELECTRICAL INTERFACE

ELECTRICAL INTERFACE

A mass storage ATA PC Card uses the PCMCIA PC Card electrical interface. A subset of the entire PCMCIA interface is sufficient for PC Card ATA implementation. Both mandatory and optional signals are given in this section. Special consideration is given to some signals whose definition is expanded when used in an ATA PC Card.

3.1 Pin Assignment Table

The following is the recommended pin assignment table for implementing ATA protocol conforming to the PCMCIA Release 2.01. The mandatory Interface signals are required for using the card in the mandatory Card Decoded and Host Decoded I/O spaces.

Table 3-1: PC Card ATA Signal Names and Pin Assignment

Pin #	PCMCIA Memory Interface Signal	PCMCIA I/O Interface Signal	PC Card ATA Mandatory Signal	PC Card ATA Optional Signal	Notes
					8
1	GND	GND	GND		
2	D3	D3	D3		
3	D4	D4	D4		
4	D5	D5	D5		
5	D6	D6	D6		
6	D7	D7	D7		
7	-CE1	-CE1	-CE1		
8	A10	A10		A10	1
9	-OE	-OE	-OE		
10	A11	A11		A11	2
11	A9	A9	A9		
12	A8	A8	A8		
13	A13	A13		A13	2
14	A14	A14		A14	2
15	-WE/PGM	-WE/PGM	-WE		
16	RDY/-BSY	-IREQ	RDY/-BSY : -IREQ		7
17	Vcc	Vcc	Vcc		
18	Vpp1	Vpp1	Vpp1 or No Connect		3
19	A16	A16		A16	2
20	A15	A15		A15	2
21	A12	A12		A12	2
22	A7	A7	A7		
23	A6	A6	A6		
24	A5	A5	A5		
35	GND	GND	GND		
36	-CD1	-CD1	-CD1		
37	D11	D11	D11		
38	D12	D12	D12		
39	D13	D13	D13		
40	D14	D14	D14		
41	D15	D15	D15		
42	-CE2	-CE2	-CE2		
43	RFSH	RFSH		RFSH	6
44		-IORD	-IORD		
45		-IOWR	-IOWR		
46	A17	A17		A17	2
47	A18	A18		A18	2
48	A19	A19		A19	2
49	A20	A20		A20	2
50	A21	A21		A21	2
51	Vcc	Vcc	Vcc		
52	Vpp2	Vpp2	Vpp2 or No Connect		3
53	A22	A22		A22	2
54	A23	A23		A23	2
55	A24	A24		A24	2
56	A25	A25		A25	2
57	RFU	RFU	No Connect		
58	RESET	RESET	RESET		

Pin #	PCMCIA Memory Interface Signal	PCMCIA I/O Interface Signal	PC Card ATA Mandatory Signal	PC Card ATA Optional Signal	Notes
					8
25	A4	A4	A4		
26	A3	A3	A3		
27	A2	A2	A2		
28	A1	A1	A1		
29	A0	A0	A0		
30	D0	D0	D0		
31	D1	D1	D1		
33	D2	D2	D2		
34	WP	-IOIS16	WP : -IOIS16		
34	GND	GND	GND		

Pin #	PCMCIA Memory Interface Signal	PCMCIA I/O Interface Signal	PC Card ATA Mandatory Signal	PC Card ATA Optional Signal	Notes
					8
59	-WAIT	-WAIT	-WAIT		
60		-INPACK	-INPACK		
61	-REG	-REG	-REG		
62	BVD2	-SPKR	Logic High unless BVD2 : -SPKR	BVD2 : -SPKR	4
63	BVD1	-STSCHG	Logic High unless BVD1 : -SPKR	BVD1 : -STSCHG	5 ~~4~~
64	D8	D8	D8		
64	D9	D9	D9		
66	D10	D10	D10		
67	-CD2	-CD2	-CD2		
68	GND	GND	GND		

General and specific notes for table 3.1 follow below.

Signal names in the PC Card ATA columns indicate dual function signals by listing the Memory Interfaced function, followed by a colon (:), and then the I/O Interfaced function of the signal. Signals in the PC Card ATA Optional column may be mandatory for particular features which the card vendor may choose to implement. The use of optional signals is described in the following numbered notes.

1. Address line A10 is mandatory if Memory Mapped addressing is supported. Otherwise, A10 is permitted to be implemented at the discretion of the card vendor.
2. Address lines A11 through A25 are permitted to be implemented at the discretion of the card vendor.
3. The use of the Vpp1 and Vpp2 supplies is optional. If they are used, it is recommended that the card vendor select +12 Volts as the Vpp value. A card which does not require any Vpp supply, shall leave both Vpp pins unconnected at the card.

   The Vpp1 and Vpp2 supplies shall not be connected to each other on the card.

   When only one supply is required, it is recommended that Vpp1 be used.

   At power up, the host shall provide at least minimal current at +5 Volts on both Vpp1 and Vpp2.
4. The -SPKR signal is optional. If the function is not implemented, this pin shall be held at logic high (negated) by the drive.
5. The -STSCHG signal is optional, however it shall be implemented if both the Card Configuration and Status Register and the Pin Replacement Register are implemented. If the function is not implemented, this pin shall be held high (negated) by the drive.

6. Note that the use of the Refresh pin has not yet been fully defined. This pin is not required for operation of the PC Card ATA peripheral.
7. The asserted state of the Ready/-Busy signal shall be interpreted to be the Busy state of the signal. See section 3.3 for additional information on the Ready/-Busy signal.
8. All signals shown in the PC Card ATA mandatory column shall be implemented by all mass storage ATA PC Cards.

3.2 Reset Conditions

There are four distinct reset conditions associated with an ATA PC Card. They are as follows:

a) Card detected Power-On Reset;

b) Host generated PCMCIA Hardware Reset using the Reset signal;

c) Host initiated PCMCIA Soft Reset using SRESET bit in the Configuration Option Register;

d) ATA Soft Reset using the SRST bit in the ATA Device Control Register.

The host should always accompany a card Power-On event with a host generated hardware reset.

Power-On, PCMCIA Hardware Reset and PCMCIA Soft Reset all place the card into the memory interfaced configuration (configuration index value of 0H) and cause the card to perform the ATA Hard Reset protocol.

The PCMCIA Soft Reset has the same effect as the Host generated hardware reset with the exception that the PCMCIA Software Reset bit itself is not cleared by the assertion of Soft Reset.

The ATA soft reset does not affect the card's PCMCIA configuration, but does perform ATA Soft Reset processing as specified in the ATA document and modified in section 6.1 of this document.

3.3 Ready/-Busy Signal and RRdy/-Bsy bit

The Ready/-Busy signal is available while the card is configured to use the Memory Interface. This signal is unavailable and is replaced by the interrupt request signal, -IREQ, while the card is configured to use the I/O interface. The Ready/-Busy signal is asserted, logic low, when the card is in the Busy condition.

If the Pin Replacement register is implemented on the card, the RRdy/ -Bsy bit in the that register, is cleared when the card is busy and set when the card is ready.

The card shall be Busy under the following conditions:

a) From Power-On until the card is ready to be accessed.

b) From PCMCIA Hardware Reset until the card is ready to be accessed.

c) From PCMCIA Soft Reset until the card is ready to be accessed.

d) If a card supports the PCMCIA Power-Down Request bit in the Configuration and Status Register, then from a change in the PCMCIA Power-Down Request bit until the card has completed the requested Power-Down or Power-Up operation.

e) While the card is in a Memory interfaced configuration, whenever the Busy bit in the ATA Status Register is set.

3.4 Interrupt Request: -IREQ

The interrupt request signal from the card (-IREQ) is available only when the card is configured to use the PCMCIA I/O interface. The handling of this signal is slightly different from the handling of the ATA interrupt request signal, +IRQ.

The polarity of the PCMCIA -IREQ signal is opposite to that of the ATA +IRQ signal. The PCMCIA -IREQ signal has a mandatory PCMCIA level mode interrupt and an optional PCMCIA pulse mode interrupt. The pulse mode interrupt is designed to allow sharing of interrupts in hosts which use an ISA compatible system bus between the PCMCIA socket and the host's CPU. To take advantage of a PCMCIA pulse mode interrupt, the host socket must be able to pass the interrupt request signal without inversion from the Card to the internal ISA bus and to drive the ISA bus +IRQn signal with an open collector driver.

When the nIEN bit in the ATA Device Control register is set, an ATA PC Card shall not assert the -IREQ signal. This is in contrast to the ATA document which specifies that the ATA interrupt request signal, +IRQ, is placed in high impedance during these times.

SECTION - 4

ATA SPECIFIC REGISTER DEFINITIONS

ATA SPECIFIC REGISTER DEFINITIONS

4.1 PC Card ATA Drive register and protocol definitions

ATA PC Cards can be configured as a high performance I/O device through standard I/O address spaces: 1F0H-1F7H, 3F6H-3F7H (primary); 170H-177H, 376H-377H (secondary) and IREQ 14 or anywhere in the I/O space or memory space requiring a dedicated driver. The communications to and from the drive is performed using the ATA Command Block which provide all the necessary control and status information. The PCMCIA interface connects peripherals to the host using four register mapping methods. Table 4.1 is a description of these methods:

Table 4-1: Standard Configurations

Config Index	IO or Memory	Address A10-A0	Drive Number	Socket & Copy	Mandatory or Optional	Description
0H[1]	Memory	0H-0FH, 400-7FFH	0	X000XXXX	Optional	Memory Mapped
1H[1]	I/O	XX0H-XXFH	0	X000XXXX	Mandatory	I/O Mapped 16 Contiguous Registers
2H[1]	I/O	1F0H-1F7H, 3F6H-3F7H ------------- 9F0H-9F7H, BF6H-BF7H	0	X000XXXX	Mandatory	Primary I/O Mapped Drive 0
2H[1]	I/O	1F0H-1F7H, 3F6H-3F7H ------------- 9F0H-9F7H, BF6H-BF7H	1	X001XXXX	Optional	Primary I/O Mapped Drive 1
3H[1]	I/O	170H-177H, 376H-377H ------------- 970H-977H, B76H-B77H	0	X000XXXX	Mandatory	Secondary I/O Mapped Drive 0
3H[1]	I/O	170H-177H, 376H-377H ------------- 970H-977H, B76H-B77H	1	X001XXXX	Optional	Secondary I/O Mapped Drive 1

[1] The configuration indices indicated here are for example only.

The host selects the card's register mapping configuration by writing the Configuration Index value to the least significant 6 bits of the card's Configuration Option Register. The actual configuration index values used by a card are vendor specific and are reported to the host using the Configuration Table Entry tuples. However, configuration 0H shall always select the PCMCIA Memory Only interface.

4.2 ATA Registers

The ATA registers are the registers which are provided on the card specifically to implement the ATA aspects of the PC Card ATA protocol. The first eight registers and duplicates are referred to as the ATA Command Block.

In accordance with the PCMCIA specification each of the registers below which is located at an odd address may be accessed using either data bus lines D15 through D8 or using data bus lines D7 through D0. Refer to items g) and h) in section 2.2 or to the PCMCIA PC Card document for more information.

4.2.1 Data Register

The Data Register is a 16-bit register which is used to transfer data blocks between the card data buffer and the Host. Data may be transferred by either a series of word accesses to the data register or a series of byte accesses to the data register. The ATA document specification sections "6.3.5 DD0-DD15" and "9.21 Set Features" specify under what conditions word and byte accesses from the host are appropriate to access this register.

Table 4-2: Data Register

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Data Word															
								Data Byte							

Refer to the ATA document for detailed information about this register. Refer also to Section 4.2.13 for additional information about the Data register, the Duplicate Data registers, and the interactions between the Data register and the Error or Feature register.

4.2.2 Error Register

This register contains additional information about the source of an error which has occurred in processing of the preceding command. This register should be checked by the host when bit 0 (ERR) of the Status register is found to be set. This register cannot be written by the host.

The bits are labeled as follows:

Table 4-3: Error Register

D7	D6	D5	D4	D3	D2	D1	D0
BBK	UNC	MC	IDNF	MCR	ABRT	TKNOF	AMNF

All bits in this register are defined in the ATA document. Refer to the ATA document for a detailed description of this register. Refer also to Section 4.2.13 for additional information about the Error register, the Duplicate Error register, and the interactions between the Data register and the Error register.

4.2.3 Feature Register

This register is written by the host to provide command specific information to the drive regarding features of the drive which the host wishes to utilize. This register cannot be read by the host and may be ignored by some drives.

Table 4-4: Feature Register

D7	D6	D5	D4	D3	D2	D1	D0
Feature Byte							

Refer to the ATA document for a detailed description of this register. Refer also to Section 4.2.13 for additional information about the Feature register, the Duplicate Feature register, and the interactions between the Data registers and the Feature register.

4.2.4 Sector Count Register

This register is written by the host with the number sectors or blocks to be processed in the subsequent command. After the command is complete, the host may read this register to obtain the count of sectors left unprocessed by the command.

Table 4-5: Sector Count Register

D7	D6	D5	D4	D3	D2	D1	D0
Sector Count							

Refer to the ATA document for a detailed description of this register.

4.2.5 Sector Number Register

This register is written by the host with the starting sector number to be used in the subsequent Cylinder-Head-Sector command. After the command is complete, the host may read the final sector number from this register. When logical block addressing is used, this register is written by the host with bits 7 to 0 of the starting logical block number and contains bits 7 to 0 of the final logical block number after the command is complete.

Table 4-6: Sector Number Register

D7	D6	D5	D4	D3	D2	D1	D0
Sector Number (CHS Addressing)							
Logical Block Number bits A07-A00 (LBA Addressing)							

Refer to the ATA document for a detailed description of this register for Cylinder-Head-Sector commands.

4.2.6 Cylinder Low Register

This register is written by the host with the low-order byte of the starting cylinder address to be used in the subsequent Cylinder-Head-Sector command. After the command is complete, the host may read the low-order byte of the final cylinder number from this register. When logical block addressing is used, this register is written by the host with bits 15 to 8 of the starting logical block number and contains bits 15 to 8 of the final logical block number after the command is complete.

Table 4-7: Cylinder Low Register

D7	D6	D5	D4	D3	D2	D1	D0
Cylinder Number Low Byte (CHS Addressing)							
Logical Block Number bits A15-A08 (LBA Addressing)							

Refer to the ATA document for a detailed description of this register for Cylinder-Head-Sector commands.

4.2.7 Cylinder High Register

This register is written by the host with the high-order byte of the starting cylinder address to be used in the subsequent Cylinder-Head-Sector command. After the command is complete, the host may read the high-order byte of the final cylinder number from this register. When logical block addressing is used, this register is written by the host with bits 23 to 16 of the starting logical block number and contains bits 23 to 16 of the final logical block number after the command is complete.

Table 4-8: Cylinder High Register

D7	D6	D5	D4	D3	D2	D1	D0
Cylinder Number High Byte (CHS Addressing)							
Logical Block Number bits A23-A16 (LBA Addressing)							

Refer to the ATA document for a detailed description of this register for Cylinder-Head-Sector commands.

4.2.8 Drive-Head Register

The Drive-Head register is used to specify the selected drive of pair of drives sharing a set of registers. The bits are defined as follows:

Table 4-9: Drive-Head Register

D7	D6	D5	D4	D3	D2	D1	D0
1	LBA (0)	1	DRV	HS3	HS2	HS1	HS0
	LBA (1)			LBA27	LBA26	LBA25	LBA24

Bit 7	1	This bit is '1'.
Bit 6	LBA	This bit is '0' for Cylinder-Head-Sector addressing and '1' for Logical Block Addressing.
Bit 5	1	This bit is '1'.
Bit 4	DRV	This bit is number of the drive which the host has selected. When DRV is cleared, drive 0 (card 0) is selected. When DRV is set, drive 1 (card 1) is selected. The card is selected to be Card 0 or to be Card 1 using the "Copy" field of the PCMCIA "Socket and Copy" configuration register, if present. If no Socket and Copy configuration register is present on the card, or if the Card's CIS indicates that it does not support twin-cards for the selected configuration, then DRV shall be cleared by the host.
Bit 3	HS3/LBA27	This is bit 3 of the head number in CHS addressing and bit 27 of the Logical Block Number in LBA addressing.
Bit 2	HS2/LBA26	This is bit 2 of the head number in CHS addressing and bit 26 of the Logical Block Number in LBA addressing.
Bit 1	HS1/LBA25	This is bit 1 of the head number in CHS addressing and bit 25 of the Logical Block Number in LBA addressing.
Bit 0	HS0/LBA24	This is bit 0 of the head number in CHS addressing and bit 24 of the Logical Block Number in LBA addressing

4.2.9 Status Register

See the description of Alternate Status Register.

4.2.10 Alternate Status Register

The Status register and the Alternate Status register return the card status when read by the host. Reading the Status register clears a pending interrupt request while reading the Auxiliary Status register does not. The status bits are identified as follows:

Table 4-10: Status and Alternate Status Register

D7	D6	D5	D4	D3	D2	D1	D0
BSY	DRDY	DWF	DSC	DRQ	CORR	IDX	ERR

Refer to the ATA document for a description of the bits which are described in this register except where the ATA document conflicts with the descriptions below.

Bit 1	IDX	This bit is optional. If implemented it shall be implemented as described in the ATA document.
Bit 5	DWF	This bit is used to indicate a drive write failure. Drives which require Vpp for write operations should use this bit to signal if the Vpp voltage is out of tolerance when a write is attempted.

4.2.11 Device Control Register

This register is used to control the card interrupt request and to issue a soft reset to the card. The bits are defined as follows:

Table 4-11: Device Control Register

D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	1	SRST	nIEN	0

Refer to the ATA document for the general description of the bits which are described in this register except where it conflicts with the descriptions noted below:

Bit 1	nIEN	While the card is operating in the memory mapped mode this bit is permitted to be ignored; but see additional requirements in Section 7.1 for cards which implement the Card Configuration and Status Register. While this bit is cleared, interrupts shall operate as described in the PCMCIA PC Card document in response to the events described in the ATA document. While this bit is set, the interrupts on the card shall be disabled. The -IREQ signal in the PCMCIA I/O interface shall be negated unless the nIEN bit is set and an interrupt has been requested.
Bit 2	SRST	The Software Reset bit shall operate generally as described in the ATA document with the following exceptions: Sections of ATA specification which refer to the PDIAG and the DASP signals are not applicable to PCMCIA implementations. Section 6.1 of this document shall define the Soft Reset Function and protocol.

4.2.12 Drive Address Register

This register is provided for compatibility with the AT disk drive interface. The bits can be read by the host and are defined as follows:

Table 4-12: Drive Address Register

D7	D6	D5	D4	D3	D2	D1	D0
X	nWTG	nHS3	nHS2	nHS1	nHS0	nDS1	nDS0

Bit 7 X This bit shall be ignored by the host. Please see Appendix B of this document for a discussion of the considerations involving this bit.

Bit 6 nWTG This bit is cleared while a write operation is in progress, otherwise, it is set.

When the bit is cleared the host should not alter the Vpp or Vcc supply voltages to the card.

Refer to ATA document for description of the bits which are described in this register except where the ATA document conflicts with the descriptions provided above.

4.2.13 Duplicate Data, Error and Feature Registers

The address space occupied by the Data register overlaps with space occupied by the Error and Feature registers. The table below describes the combinations of Data register access and Error or Feature register accesses. The table is provided here to assist in understanding the overlapped Data register and Error or Feature register rather than to attempt to define general PCMCIA word and byte access modes and operations. See the PCMCIA PC Card document for definitions of the Card Accessing Modes for I/O and Memory cycles. These cycles are also summarized in section 2.2, items g, h, and i.

Table 4-13: Duplicate Data Register

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Data Word															
Odd Data Byte Only								Even or Even-Odd Data Byte							

Because of the overlapped registers, access to the Error or Feature register at 1F1H, 171H and offset 1h are not possible when word accesses are performed, i.e. with -CE2 and -CE1 both asserted. The duplicate registers at relative addresses 8H, 9H and 0DH have no restrictions on the operations which can be performed by the socket.

Table 4-14: Access to Data, Error and Feature Registers Including Duplicate Registers

Data Register	-CE2	-CE1	A0	Offset	Data Bus
Word Data Register	0	0	0	0H, 8H	D15-D0
Word Data Register	0	0	1	1H, 9H	D15-D0
Even Byte Data Register	1	0	0	0H,8H	D7-D0
Odd Byte Data Register	1	0	1	9H	D7-D0
Odd Byte Data Register	0	1	X	8H, 9H	D15-D8
Error / Feature Register	1	0	1	1H, 0DH	D7-D0
Error / Feature Register	0	1	X	0H, 1H	D15-D8
Error / Feature Register	0	0	X	0C, 0DH	D15-D8

Notes:

1. The Data register at 0h is accessed with both -CE1 and -CE2 asserted as a word register on the combined Odd Data Bus and Even Data Bus (D15-D0). This register may also be accessed by a pair of byte accesses to the offset 0H with -CE1 asserted and -CE2 negated. Word accesses at odd address N+1 is the same as a word access at address N, however, word accesses at odd addresses are illegal for I/O accesses. Note that the address space of this word register overlaps the address space of the Error and Feature byte-wide registers which are located at offset 1H. When accessed twice as byte register with -CE1 asserted, the first byte to be accessed is the Even byte of the Word and the second byte accessed is the Odd byte of the equivalent Word access.

   A byte access to register 0H with -CE1 negated and -CE2 asserted accesses the Error (read) or Feature (write) register.

2. The registers located at offsets 8H,9H and 0DH are non-overlapping duplicates of the registers at offsets 0 and 1.

   Register 8H is equivalent to register 0H, while register 9H accesses only the Odd byte of the Data register. Therefore, if the registers are byte accessed in the order 9H then 8H the data will be transferred Odd byte then Even byte. Repeated byte accesses to register 8H or 0H will access consecutive (even then odd) bytes from the data buffer. Repeated word accesses to register 8H, 9H or 0H will access consecutive words from the data buffer. Repeated byte accesses to register 9H are not supported. However, repeated alternating byte accesses to registers 8H then 9H will access consecutive (even then odd) bytes from the data buffer. Byte accesses to register 9H access only the odd byte of the data word.

3. Memory accesses to even addresses at offsets between 400H and 7FFH access register 8H. Accesses to odd addresses at offsets between 400H and 7FFH access register 9H. This 1 Kbyte memory window to the data register is provided so that hosts can perform memory to memory block moves to the data register when the register lies in memory space. Note that this entire window accesses the Data Register FIFO and does not directly address the data buffer within the card.

   Some hosts, such as the 80x86 processors, increment both the source and destination addresses when executing the memory to memory block move instruction. Some PCMCIA socket adapters also have auto incrementing address logic embedded within them. This address window allows these hosts and adapters to function efficiently.

4.3 ATA Specific Register Mapping

4.3.1 I/O Mapped Addressing

The Primary I/O, Secondary I/O, and Contiguous I/O address maps are shown in Table 4-15.

The contiguous I/O mapping mode requires that the system decode a contiguous block of at least 16 I/O registers to uniquely select the card.

Table 4-15: I/O Mapped Addressing

-REG	Primary A9-A0	Secondary A9-A0	Contiguous A3-A0	-IORD=0	-IOWR =0	Note
0	$1F0_H$	170_H	0_H	Even Read Data	Even Write Data	1,2
0	$1F1_H$	171_H	1_H	Error Register	Features	2
0	$1F2_H$	172_H	2_H	Sector Count	Sector Count	
0	$1F3_H$	173_H	3_H	Sector Number	Sector Number	
0	$1F4_H$	174_H	4_H	Cylinder Low	Cylinder Low	
0	$1F5_H$	175_H	5_H	Cylinder High	Cylinder High	
0	$1F6_H$	176_H	6_H	Drive/Head	Drive / Head	
0	$1F7_H$	177_H	7_H	Status	Command	
0	---	---	8_H	Duplicate Even Read Data	Duplicate Even Write Data	1,3
0	---	---	9_H	Duplicate Odd Read Data	Duplicate Odd Write Data	3
0	---	---	$0D_H$	Duplicate Error	Duplicate Features	3
0	$3F6_H$	376_H	$0E_H$	Alternate Status	Device Control	
0	$3F7_H$	377_H	$0F_H$	Drive Address	Reserved	

Address lines which are not indicated in the decoding above are ignored by the card for accessing these registers. The primary and secondary modes decode 10 address lines while the contiguous decoding decodes only 4 address lines on the card.

Note 1: This register supports word or byte accesses. See the corresponding note for table 4-14.

Note 2: This register overlaps the address space of the Data Register. See the corresponding note for table 4-14.

Note 3: This register address is a duplicate address assignment for another register. A duplicate address is not available in the Primary I/O and Secondary I/O decodings. See the corresponding note for table 4-14.

4.3.2 Memory Mapped Addressing

When the card registers are accessed via memory references, the registers appear in the common memory space window from 0-2K bytes as shown in Table 4-16.

Table 4-16: Memory Mapped Address Map

-REG	A10	A9-A4	A3	A2	A1	A0	-OE=0	-WE=0	Notes
1	0	X	0	0	0	0	Read Data	Write Data	1
1	0	X	0	0	0	1	Error	Features	2
1	0	X	0	0	1	0	Sector Count	Sector Count	
1	0	X	0	0	1	1	Sector Number	Sector Number	
1	0	X	0	1	0	0	Cylinder Low	Cylinder Low	
1	0	X	0	1	0	1	Cylinder High	Cylinder High	
1	0	X	0	1	1	0	Select Card /Head	Select Card/Head	
1	0	X	0	1	1	1	Status	Command	
1	0	X	1	0	0	0	Duplicate Even Read Data	Duplicate Even Write Data	2
1	0	X	1	0	0	1	Duplicate Odd Read Data	Duplicate Odd Write Data	2
1	0	X	1	1	0	1	Duplicate Error	Duplicate Features	2
1	0	X	1	1	1	0	Alt Status	Device Ctl	
1	0	X	1	1	1	1	Drive Address	Reserved	
1	1	X	X	X	X	0	Even Read Data	Even Write Data	3
1	1	X	X	X	X	1	Odd Read Data	Odd Write Data	3

Notes: 1,2,3 See the corresponding note for Table 4-14.

If memory mapped mode is supported, the card shall be implemented so that the card will respond at the addresses indicated within the ranges of 0H-0FH and 400H-7FFH from the start of the address space allocated to the PC Card ATA memory mapped registers as indicated by the CIS Device ID and JEDEC ID tuples. Additional decoding is permitted to be provided at the discretion of the card manufacturer.

SECTION - 5

SOFTWARE INTERFACE

SOFTWARE INTERFACE

This section defines the software requirements and the commands the host sends to the Storage System. The controller executes the commands and reports the results to the host using the Status byte.

5.1 ATA Command Block

The ATA command block is the group of seven registers which are used to issue commands using the ATA command protocol. The interpretation of the contents of these registers is a function of the addressing mode which is used to address the media in the card. A Cylinder-Head-Sector addressing method of addressing and a new Logical Block addressing mode are supported.

5.1.1 ATA Command Block for Cylinder-Head-Sector Addressing

To perform a function the host writes up to seven bytes to the card. These bytes, called the ATA Command Block, specifies the command to be executed and its associated parameters. The following figure shows the general content of the ATA command block. Refer to specific commands in the ATA document for the bytes required by each command.

Table 5-1: Commands with Cylinder-Head-Sector encoding:

Bit →	7	6	5	4	3	2	1	0
(Byte offset)1	Features							
2	Sector Count							
3	Sector Number							
4	Cylinder Low							
5	Cylinder High							
6	1	LBA=0	1	DRV	Head			
7	Command							

5.1.2 ATA Command Block for Logical Block Addressing

To perform a function using Logic Block Addressing, the host writes to the same seven registers as for Cylinder-Head-Sector addressing. However, the LBA bit is Set and the Sector Number, Cylinder Low, Cylinder High and Head fields of the command block provide a starting logical block address on the card. They are interpreted as follows:

Table 5-2: Commands with Logical Block Address encoding

Bit →	7	6	5	4	3	2	1	0
(Byte offset) 1	Features							
2	Sector Count							
3	Logical Block Number A7-A0							
4	Logical Block Number A15-A8							
5	Logical Block Number A23-A16							
6	1	**LBA=1**	1	DRV	Logical Block Number A27-A24			
7	Command							

CHS to LBA translation formula: $LBA = (C * HpC + H) * SpH + S - 1$

LBA to CHS translation formulas: $C = LBA \ / \ (HpC * SpH)$
$H = (LBA \ / \ SpH) \bmod HpC$
$S = (LBA \bmod SpH) + 1$

where	LBA	is Logical Block Address
	C	is Cylinder Number
	H	s Head Number
	S	is Sector Number
	HpC	is Heads per Cylinder
	SpH	is Sectors per Head (Track)

5.2 Command Descriptions

The ATA document should be consulted for detailed tables and descriptions of the commands.

Command descriptions are presented in the ATA document utilizing Cylinder-Head-Sector parameters.

SECTION - 6

INTERFACE PROTOCOL

INTERFACE PROTOCOL

Refer to section 7 in the ATA document for Logical Interface descriptions with the following exceptions:

a. All references to the PDIAG and the DASP signals shall be ignored.

b. Port addressing given in the ATA Document in Table 7-1 is replaced by sections 4.3.1 and 4.3.2 of this document.

c. The high impedance state of INTRQ is replaced by the interrupts disabled state as described in section 3.4 of this document.

d. Support for the Index bit (IDX) in the status registers is optional.

e. The host provides drive number configuration of each card by writing bit 4 of the Card's Socket and Copy register with the value 0 for drive 0 or with the value 1 for drive 1.

f. Within the protocol overview provided in section 10 of the ATA document, Sections 10.5 through 10.5.3 do not apply because DMA is not supported by the PCMCIA interface.

g. Within section 11 of the ATA Document describing timing protocols, only sections 11.1 through 11.3 apply to ATA PC Cards. Section 11.4, Data transfer timing, is replaced by the corresponding information from the PCMCIA PC Card Document.

6.1 ATA Soft Reset

This bit is set to 1 in the Device Control register to force the card to perform an AT Disk Controller hard reset operation. This reset does not change the configuration of the card interface as would either a PCMCIA hardware reset or a PCMCIA Soft Reset.

When drive 0 is ready drive 1 is not necessarily ready. Therefore, an attempt to access drive 1 will fail, because PCMCIA specification does not allow inter-drive communication, and attempt to read Status register on drive 1 will fail because the drive bit cannot be written while the drive is busy. In order to avoid a potential problem the following protocol comparable to section A.2 of the ATA document should be used:

6.1.1 ATA Soft Reset Timing Definitions

Definitions: tB Drive 0 is the time from ATA Soft Reset cleared until drive 0 clears BSY when Drive 1 may be present. It is specified as a minimum.

tB Drive 1 is the time from ATA Soft Reset cleared until drive 1 clears BSY. It is specified as a maximum.

tN is the time from ATA Soft Reset set until the drive sets BSY. It is specified as a maximum.

tU is the time from the posting of Drive Ready and Diagnostic Results until the drive clears BSY. It is specified as a minimum.

6.1.2 Software Reset One Drive

This protocol applies only when the drive is configured so that it is the only drive which can be present at an address. This is the case when an I/O Mapped Configuration without Twin Cards support or the memory mapped configuration is used.

1. Host sets SRST=1 in the Device Control Register.

2. Drive 0 sets BSY within 400 nsec after SRST is set to 1.
3. Drive 0 begins hardware initialization.
4. Drive 0 may revert to its default condition.
5. Drive 0 posts diagnostic results to the Error Register.
6. Drive 0 clears BSY when ready to accept commands.

6.1.3 Software Reset Two Drives

This protocol applies whenever the drive is in a configuration which supports more than one drive at a single address whether or not more than one drive is actually present at that address in the system.

1. Host sets SRST=1 in the Device Control Register.
2. Drive 0 and Drive 1 each set BSY within 400 nsec after SRST is set.
3. Drive 0 and Drive 1 begin hardware initialization.
4. Drive 0 and Drive 1 may revert to their default condition.

Drive 1

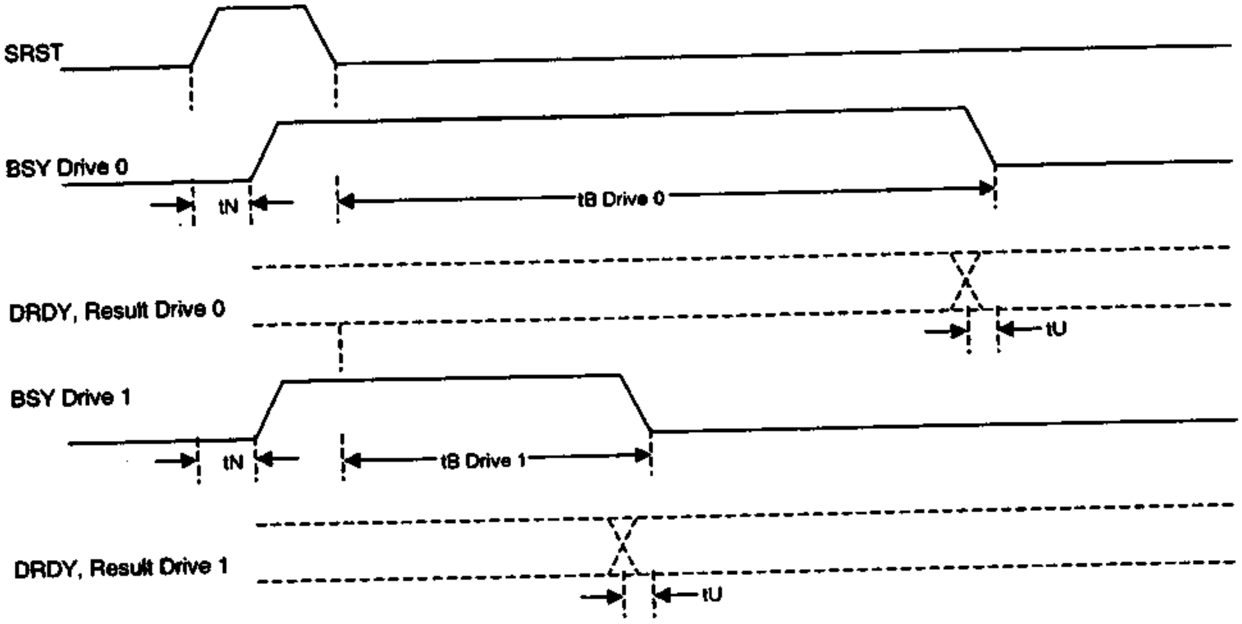
5. Drive 1 performs initialization and diagnostics which will complete within tB Drive 1 after Soft Reset is cleared.
6. With adequate time remaining to complete steps 7 through 9 before tB Drive 1 has expired, Drive 1 determines whether it has completed diagnostics and is ready to execute a command.
7. If the diagnostic results are uncertain at this time, then the drive will report the diagnostics as having passed. The drive shall place the value "01H" (passed diagnostic) in the Error Register.

   If the diagnostic results are available at this time, then the drive shall place the diagnostic result in the Error register.
8. If the drive is ready to execute a command then the drive shall set the DRDY bit in the Status register otherwise the drive shall clear the DRDY bit in the Status register.
9. The drive clears BSY in the Status register.

Drive 0

5. Drive 0 shall perform and complete initialization and diagnostics.
6. Drive 0 shall delay until tB Drive 0 has expired.
7. Drive shall place the diagnostic result in the Error register with the assumption that drive 1 has passed diagnostics.
8. If the drive is ready to execute a command then the drive shall set the DRDY bit in the Status register otherwise the drive shall clear the DRDY bit in the Status register.
9. The drive clears BSY in the Status register.

Table 6-1: Soft Reset Timing Diagram



Label	Value	Units		Conditions
tN	400	nsec	Max	All
tB Drive 0	0	msec	Min	Single Drive Configuration
tB Drive 0	100	msec	Min	Multi Drive Configuration
tB Drive 1	50	msec	Max	Multi Drive Configuration
tU	0	nsec	Min	All

SECTION - 7

PCMCIA SPECIFIC CONSIDERATIONS

PCMCIA SPECIFIC CONSIDERATIONS

7.1 Card Configuration Registers

The PCMCIA Card Configuration Registers are described in the PCMCIA PC Card Standard Release 2.01 section 4.15. When only some of the bits in a register are required by a card, the unsupported bits may be ignored when written and should return stable data (typically 0) when read.

The Configuration Option Register is the only register which is mandatory for all ATA PC Cards. This register is used to specify the addressing mode of the card, the interrupt mode (Level or Pulsed) and to assert PCMCIA Soft Reset.

The Card Configuration and Status Register is required to be implemented on the card only if the card supports the PCMCIA power-down, audio, or Signal on Change features, none of which are mandatory. If PCMCIA power-down is supported, it is recommended that this place the drive in the lowest power state available from which the drive can recover by restoring the PCMCIA Power Down Bit to 0. The Signal on Change feature also requires that the Pin Replacement Register be implemented.

If the Card Configuration and Status Register is implemented on the card the Interrupt Request bit in the register shall be controlled as follows: The bit shall be set when the drive has an interrupt request pending and the Interrupt Enable bit in the ATA Device Control Register is set to permit interrupts. While a drive is configured for the memory-only interface, the behavior of the Interrupt Request bit in the Card Configuration and Status Register is the same as if an I/O interface were configured, although the Hardware Interrupt Request signal will not be available from the card.

The IOis8-bit in the Card Configuration and Status Register is set by the host to inform the card whether the host will perform all I/O to the card as 8-bit I/O transferred on the Even Data Bus. This bit should not be interpreted as controlling the width of data accesses to the card.

The Pin Replacement Register is required to be implemented on the card only if the card is designed to return Ready/Busy, Write Protect or Battery Status while the card is using the I/O interface.

The Socket and Copy Register, bit 4, is required to be implemented on the card if the card is designed to be host selectable as either drive 0 or as drive 1. A card indicates this capability with the Twin Cards field in the Configuration Entry Tuple of the Card Information Structure. The drive number selection is performed by clearing bit 4 of the Socket and Copy Register to 0 for drive 0 or setting the bit to 1 for drive 1. The twin card operation is intended for use in the AT Primary and AT Secondary I/O mapped addressing Configurations.

7.2 Card Removal, Insertion and Change Detection

The ATA Card insertion and removal shall be detected by having the socket monitor the Card Detect pins of the card and notify the client driver when there is a change in their status. The Identify drive command can be used (Model and Serial Number) to determine whether a drive which is inserted into a socket is already mounted by the system. Be aware, however, that the data on the PC card may have been altered on another system between the time it is removed from and then replaced back into the first system.

APPENDIX - A

GLOSSARY

GLOSSARY

This section is intended to give general descriptions of terms used in this document. In many cases, more detailed information about these terms may be found in the PCMCIA PC Card Document or the ATA Document.

Address Space .... An address space is a collection of registers and storage locations contained on a PC Card which are distinguished from each other by the value of the Address Lines applied to the Card. There are three, separate, address spaces possible for a card. These are the Common Memory space, the Attribute Memory space and the I/O space.

Asserted.. A signal is asserted when it is in the state which is indicated by the name of the signal. See the Conventions section. Opposite of Negated.

AT.. Acronym for Advanced Technology. Refers to a 16 bit Personal Computer architecture using the 80x86 processor family which formed the basis for the ISA Bus definition.

ATA.. An acronym for A T Attachment. Refers to the interface and protocol used to access a hard disk in AT compatible computers. Disk drives adhering to the ATA protocol are commonly referred to as IDE interfaced drives for PC compatible computers.

ATA Command Block................ See Command Block Registers.

ATA Document........................... The latest draft of the ANSI X3.T9 subcommittee AT Attachment document. See the related documents section.

ATA Registers .... These registers are accessed by a host to implement the ATA protocol for transferring data, control and status information to and from the PC Card. They are defined in the ATA Document. These registers include the Cylinder High, Cylinder Low, Sector Number, Sector Count, Drive-Head, Drive Address, Device Control, Error, Feature, Status and Data registers. The I/O and memory address decoding options for these registers are defined within this specification.

ATA Soft Reset .... The condition of the ATA PC Card when the SRST bit in the Device Control Register is set. This condition directly affects only the ATA Registers and protocol. Except for reflecting the state of the Ready/-Busy function, the Configuration Registers are unaffected.

Attribute Memory Space............ One of the three address spaces available on a PCMCIA PC Card. This address space is accessed by memory read and memory write operations which occur while the -REG signal is asserted. This address space is defined only for bytes located on even byte addresses. This space is the primary location for the Card Information Structure and for the Configuration Registers on the card.

Block... A block is the basic 512 byte region of storage into which the storage media is divided. Addressing in the ATA protocol is performed on block boundaries. Each block of data represents one sector of data using the ATA Cylinder-Head-Sector address protocol. The Logical Block Address protocol uses sequential block addresses to access the media.

BSY (ATA Busy bit) .... A bit in the ATA Status Register which is used by the ATA protocol to indicate that the ATA registers on the card are not available for use by the host.

Card Configuration and Status Register.. This Configuration Register is located 2 bytes above the Configuration Option Register. It provides the host control for the following functions: Status Changed Signal, Audio Signal, and Power Down Request. It provides status information about Status Changed State, Interrupt Request State. In addition it can be used to advise the card that all I/O to the card will be eight bits wide. Refer to the PC Card document for detailed information about this register.

Card Enumeration .... The process performed by the host to provide a unique card identification number to each drive when the Twin Cards option is used. The host writes a unique number to the Copy field in the Socket and Copy register of each card sharing the same configuration.

Card Information Structure........ A data structure which is stored on a PCMCIA PC Card in a standard manner which contains information about the capabilities of the card as well as the formatting and organization of data on the card.

CHS.. An acronym for Cylinder-Head-Sector addressing.

CIS.. CIS is an acronym for the Card Information Structure.

Clear (a bit) .... A bit is Cleared when its value is set to "0."

Command Block Registers .... The ATA Command Block Registers include the following ATA registers: Data Register(s), Error Register, Feature Register, Sector Count Register, Sector Number Register, Cylinder Low Register, Drive-Head Register, Command Register, and Status Register, but not the Alternate Status Register. Seven of the Command Block Registers are written by the host to provide a command and its parameters. These registers are: Feature Register, Sector Count Register, Sector Number Register, Cylinder Low Register, Cylinder High Register, Drive-Head Register and Command Register.

Common Memory Space .... This, 16 bit wide, memory space is one of the three address spaces available on the PC Card. This address space is accessed by memory read and memory write operations which occur while the -REG signal is negated. This address space is defined only for both bytes located at even and odd byte addresses. A PCMCIA bus multiplexing protocol is used to ensure that the odd bytes of this space can be accessed by both eight and 16 bit hosts. The ATA registers are located in this space when Memory Mapped ATA registers are supported.

Configuration .... Configuration is a process by which a host initializes or alters its socket operation and the Configuration Registers on a PC card to match the PC Card's capabilities to the host's capabilities and available system resources.

Configuration Option Register .. This register is the first of the Card Configuration Registers located in the Attribute Memory Space of a PC Card. It is used by the host to control the Card's Configuration Index in bits 5 to 0, its Interrupt Mode in bit 6 (Pulsed = 0 or Level =1) and the PCMCIA Soft Reset in bit 7 (Soft Reset asserted = 1).

Configuration Registers .... A set of registers, defined by the PCMCIA PC Card Standard, which are used by the host to control the operational configuration of the card.

Contiguous I/O Addresses .... An I/O address decoding in which the Card decodes address lines A3 through A0, while the Socket is responsible for decoding all other address lines to produce the Card Enable signals for I/O cycles to the card.

Cylinder-Head-Sector
Address .... A method for specifying the location of a block of data on a mass storage device. This is the traditional method for addressing a block of data on rotating media using the ATA protocol. This method partitions an address into a cylinder portion, one or more heads within each cylinder and one or sectors of one block each within each cylinder-head combination.

Edge Sensitive Interrupt .... A host system interrupt which causes at most one interrupt for each transition of the interrupt request signal to the asserted state. Commonly used in ISA Bus machines.

EISA Bus... Acronym for Extended Industry Standard Architecture. An internal host Bus which is available in some hosts and can be used to connect PCMCIA sockets to the host CPU. While serving the same basic purpose as a Micro Channel bus or an ISA bus some bus protocols and signals are different. An EISA bus can program each interrupt request line for either positive-true, edge sensitive, interrupts (+IRQn) or negative-true, level sensitive, interrupts (-IRQn).

Field .... A collection of bits in a register or a tuple which have an effect or meaning based the value represented by entire collection of bits. The values of fields are expressed as hexadecimal values as indicated by an "H" which follows the value.

Hardware Reset............................. See PCMCIA Hardware Reset.

High (Logic Level) .... A signal is in the high logic state when it is above approximately 2.5 volts. See the PCMCIA PC Card Document for the precise electrical definition.

Host... A computer system or other equipment which contains hardware (a Socket) and software for utilizing a PC Card.

I/O .... An abbreviation for Input / Output.

I/O Address Space .... The I/O address space is one of the three address spaces available on a PC Card. The I/O address space is accessed by asserting the I/O Read signal, -IORD, or the I/O Write signal, -IOWR, while the Attribute Memory Select Signal, -REG, and at least one Card Enable, -CE1 or -CE2 is asserted.

I/O Cycle .... An I/O cycle is an Input (I/O Read) operation or Output (I/O Write) operation which accesses the PC Card's I/O address space.

I/O Interface................................ The I/O Interface is an interface supporting both memory cycles and I/O cycles. This interface is not active at power up or following a PCMCIA reset. This interface is permitted to be enabled when both the PCMCIA socket and PC Card installed in the socket support the I/O interface. The host configures a PC Card for the I/O interface using the Configuration Option Register. PC Cards which support the I/O interface must indicate their support in the CIS on the card.

I/O Mapped .... A storage location or register is I/O mapped when it is available to be accessed using I/O cycles. The register or storage location might also be accessible using memory cycles, in which case it would also be memory mapped.

-IREQ .... The Interrupt Request signal between a PC Card and a socket when the I/O interface is active.

IRQn.. One of the Interrupt Request Signals between a socket and the host's CPU. Selection of the specific Interrupt Request Signal which is used to carry an Interrupt Request from a PC Card to the Host's CPU is controlled by hardware associated with the socket. Depending upon the host system implementation the IRQn signal may be either +IRQn or -IRQn.

ISA Bus .... An acronym for Industry Standard Architecture Bus. An internal host Bus which is available in some hosts and can be used to connect PCMCIA sockets to the host CPU. While serving the same basic purpose as a Micro Channel bus or an EISA bus some bus protocols and signals are different. An ISA bus uses positive-true, edge sensitive, interrupt request lines (+IRQn).

LBA .... An Acronym for Logical Block Address. See Logical Black Address.

LSB .... An acronym for Least Significant Bit and Least Significant Byte. That portion of a number, address or field which occurs rightmost when its value is written as a single number in conventional hexadecimal or binary notation. The portion of the number having the least weight in a mathematical calculation using the value.

Level Mode Interrupt .... A method of transmitting an Interrupt Request from a PC Card to a socket using the -IREQ signal. In this mode, the -IREQ signal is asserted when the Card initiates an interrupt and is negated when the Host acknowledges to the PC Card that the interrupt has been serviced. The method of acknowledgment is specific to devices on the PC Card. In the case of an ATA PC Card, acknowledgment takes place when the ATA Status Register is read.

The socket must use an open-collector non-inverting driver when driving a Micro Channel -IRQn signal from the PC Card's Level Mode -IREQ signal. The socket must use an inverting driver when driving an ISA +IRQn signal from the PC Card's Level Mode -IREQ signal.

LBA .... See Logical Block Address.

Level Sensitive Interrupt............. A host system interrupt which causes repeated interrupts as long as the interrupt request signal is in the asserted state and the interrupt request is not disabled. Used in Micro Channel Architecture bus hosts and available in EISA bus hosts.

Logical Block Address................. A logical block address is a sequential address for accessing the blocks on the storage media. The first block of the media is addressed as block 0 and succeeding blocks are numbered sequentially until the last block is encountered. This is the traditional method for accessing peripherals on a SCSI interface bus.

Low (Logic Level) .... A signal is in the low logic level when it is below approximately 0.5 volts. See the PCMCIA PC Card Document for the precise electrical definition.

Mandatory .... A characteristic or feature which must be present in every implementation of the standard.

Micro Channel
Architecture Bus........................... An internal host Bus which is available in some hosts and can be used to connect PCMCIA sockets to the host CPU. While serving the same basic purpose as an ISA bus or an EISA bus some bus protocols and signals are different. A Micro Channel Bus uses negative-true, level sensitive, interrupt request lines (-IRQn).

Memory Cycle .... A memory cycle is a memory read (using Output Enable) operation or memory write (using Write Enable / Program) operation which accesses the PC Card's common memory or attribute memory address space.

Memory Interface.......................... The memory interface is the default interface after power up, PCMCIA Hard Reset and PCMCIA Soft Reset for both PCMCIA cards and sockets. This interface supports memory operations only. Contrast with I/O interface.

Memory Mapped .... A storage location or register is memory mapped when it is available to be accessed using memory cycles. The register or storage location might also be accessible using I/O cycles, in which case it would also be I/O mapped.

MSB... An acronym for Most Significant Bit and Most Significant Byte. That portion of a number, address or field which occurs leftmost when its value is written as a single number in conventional hexadecimal or binary notation. The portion of the number having the most weight in a mathematical calculation using the value.

Negated .... A signal is negated when it is in the state opposed to that which is indicated by the name of the signal. See the Conventions section. Opposite of Asserted.

Offset.. The offset of a port or a memory location is the difference between the address of the specific port or memory address and the address of the first port or memory address within a contiguous group of ports or a memory window. This term is used when identifying the locations of registers located with respect to the base address of the 16 contiguous I/O ports. It is also used when identifying the location of memory mapped registers with respect to the base address of the memory window.

Optional... A characteristic or feature which is not mandatory, but is specifically permitted. If an optional characteristic or feature is present, it must be implemented as described in this specification. Optional characteristics or features are specifically identified in this document.

PC .... An acronym for Personal Computer. Often used to refer to an 80x86 based computer system.

PC Card .... A PC Card is a card which conforms to the PCMCIA PC Card standard as described in the PCMCIA PC Card Document.

PCMCIA .... An acronym for the Personal Computer Memory Card International Association. This is the body which publishes this document and the PCMCIA PC Card Document.

PCMCIA PC Card Document .... The PCMCIA PC Card Standard. The applicable revision is given in the Related Documents section.

PCMCIA Hardware Reset .... PCMCIA Hardware Reset is caused when the socket asserts the Reset signal to the PC Card. The PCMCIA Hardware Reset causes the PCMCIA interface to be made the Memory Only Interface and the Configuration Option Register to be made 00H. Other configuration registers and the Ready/-Busy signal are also affected as detailed in the PCMCIA PC Card Document.

PCMCIA Soft Reset...................... PCMCIA Soft Reset is caused when the host sets bit 7 of a PC Card's Configuration Option Register. PCMCIA Soft Reset is asserted while bit 7 of Configuration Option Register is set. The effect of PCMCIA Soft Reset is identical to the effect of PCMCIA Hardware Reset except that bit 7 of the Configuration Option Register is not cleared by the reset condition. Because the other bits of the Configuration Option are written at the same time as the PCMCIA Soft Reset bit, it is recommended that the PCMCIA Soft Reset bit be cleared by writing a 00H to the Configuration Option Register.

Pin Replacement Register .... The Pin Replacement Register is the third Card Configuration Register. It is used to retrieve status information from the PC Card about Battery, Busy and Write Protect status while the card has the I/O interface active.

Primary I/O Addresses .... The Primary I/O Address for a PCMCIA Card is the set of addresses 1F0H-1F7H and 3F6H-3F7H at which the first fixed disk controller is located in a PC/AT computer system. Use of these addresses allows emulation of the first ATA or IDE disk controller at its standard addresses.

Pulse Mode Interrupt .... A method of transmitting an Interrupt Request from a PC Card to a socket using the -IREQ signal. In this mode, the -IREQ signal is asserted momentarily when the Card initiates an interrupt and is then negated regardless of whether or not the interrupt is acknowledged. The method of acknowledgment is specific to devices on the PC Card. In the case of an ATA PC Card, acknowledgment takes place when the ATA Status Register is read. The pulse mode interrupt is designed to be used with the ISA bus (and the EISA bus when ISA bus interrupt emulation is being performed.) The host socket must use an "open-collector" non-inverting output to drive the ISA bus +IRQn signal when it is expecting to share pulse mode interrupts from the PC Card.

Ready/-Busy.................................. The Ready/-Busy signal is used by a PC Card to indicate that it is busy with an internal operation and access to the card may be restricted. Refer to Section 3.3 in this document about the Ready/-Busy signal and RRdy/-Bsy bit as well as the PCMCIA PC Card document for detailed information about this signal.

RRdy/-Bsy .... The Registered Ready/-Busy Status Bit, RRdy/-Bsy, is located in the Pin Replacement Register if that register is present on the PC Card. The bit is provided to indicate the state of the Ready/-Busy function while the Ready/-Busy signal is unavailable because the Memory Only Interface is not currently configured on the card.

Secondary I/O Addresses .... The Secondary I/O Address for an ATA PC Card is the set of addresses 170H-177H and 376H-377H at which the second fixed disk controller is located in a PC/AT computer system. Use of these addresses allows emulation of the second ATA or IDE disk controller at its standard addresses.

Set (a bit)....................................... A bit is set when its value is set to "1."

Socket... The socket is the hardware in the host which is responsible for accepting a PC Card into the host and mapping the host's internal bus signals to the PCMCIA interface signals.

Socket and Copy Register........... The Socket and Copy Register is the fourth Card Configuration Register located on a PC Card. This Configuration Register allows the host to configure an ATA PC Card to respond as either Drive 0 or Drive 1.

SRST (Soft Reset Bit)..................... The ATA Soft Reset Bit, SRST, is located in the Device Control register of a PCMCIA-ATA peripheral. This bit provides the ATA Soft Reset Function but does not cause the PCMCIA interface to perform PCMCIA Reset processing.

Status Changed Signal
-STSCHG .... The Status Changed Signal is present at the PC Card interface only when the I/O Interface is enabled. It is asserted when any of the four Changed Status bits in the Pin Replacement Register are set while the Enable Status Changed bit is set in the Card Configuration and Status Register. This signal replaces the BVD1 signal of the Memory Only interface when the I/O Interface is configured.

Task File Registers .... Obsolete version of the ATA document referred to the ATA Command Block registers as the Task File. See Command Block Registers.

Tuple .... A tuple is an element of a Card Information Structure. Each tuple has a tuple code which identifies the type of tuple which is present, a tuple length which specifies the amount of space occupied by the tuple and an information area which contains the content of the tuple. Tuples located in the CIS of a PC Card are examined by host software to determine the capabilities the card.

Twin Cards .... An optional field in a Configuration Entry tuple which permits configurations to be described in which several cards share the same system resources such as I/O ports. The cards are uniquely labelled by the host using the Copy Number field of the Socket and Copy Register. This feature is used to permit a Drive 0 and a Drive 1 to coexist at the same Primary or Secondary I/O addresses. Support for the Twin Cards Option is optional in ATA PC Cards.

APPENDIX - B

IMPLEMENTATION NOTES

IMPLEMENTATION NOTES

B.1 Special Handling of I/O Ports 3F7H and 377H

The standard, AT-BIOS compatible, address for the Drive Address Register at its primary location is shared with bit 7, the Disk Change bit, of the Floppy Disk Controller at its standard primary location.

A non-PCMCIA, ATA host adapter prevents a bus conflict between the Floppy Disk Controller and the ATA peripheral by keeping data bit 7 in high-impedance at the system bus while the register is read.

When an ATA host bus adapter is used, the Floppy disk controller and the ATA Drive are connected to the same physical wires on the data bus, so that when the Drive Address register is accessed, the floppy disk controller places D6-D0 in the high impedance state while the ATA drive places D7 in the high impedance state. This action prevents a bus conflict.

A PCMCIA socket in a host is likely to include a bus transceiver between the card's data pins and the host's system data bus. Unless the socket has been custom designed to resolve this problem, the bus transceiver is unable to generate a high-impedance output on the system data bus signal D7 in response to a high impedance input from the D7 data line on the PCMCIA socket. Therefore, the traditional ATA solution to the 3F7H register is not directly usable in the PCMCIA interface.

Therefore, an ATA PC Card configured to operate at the Primary (or Secondary) I/O addresses, conflicts with a floppy disk controller which resides in the system and also uses port 3F7H (or 377H). A conflict also occurs if the bus width supported by the ATA PC Card and the floppy disk controller are not equal.

The following are methods to avoid this condition in PCMCIA implementations. The selection of the best mechanism for a particular system will depend upon the characteristics of the host's socket, the host's driver software and ATA PC Card installed in the socket.

1. Locate the ATA PC Card at a non-conflicting address. In hosts where a Floppy Disk Controller is potentially present at 3F7H in the ATA PC Card's Primary I/O address range, the ATA PC Card would not be configured to use its Primary I/O address range. Either a contiguous I/O space decoded by the socket at a in a non-conflicting area of the I/O space or the PC Card ATA's Secondary address range, 170H to 177H with 376H to 377H, would be configured by the host.

   This method will work with any socket and ATA PC Card but requires that the software which accesses the card be aware of the location of the I/O ports for the card.

2. Hosts in which it is impossible for a Floppy Disk Controller and a ATA PC Card to reside in the system at the same time are not subject to this problem.

   This method will work only in systems where it is not possible to install both devices at the same time. For example, a system with a single PCMCIA socket, no embedded Floppy Disk Controller and no I/O expansion bus.

3. Avoid enabling the ATA PC Card's Drive Address register. There are two conditions to allow this method. 1) The software used to access the ATA PC Card must not use this register. 2) The port on the ATA PC Card must be prevented by socket or card hardware from responding. This may be accomplished in two ways.

a) If byte granularity of I/O port address decoding is supported by the socket, the socket would be programmed to enable the ATA PC Card only for I/O addresses 1F0 through 1F7H and 3F6H for a Primary address conflict. For a Secondary address conflict, the socket would be programmed to enable only I/O addresses 170H through 177H and 376H to the card.

b) If the ATA PC Card provides an additional Primary or Secondary configuration of the card which does not respond to accesses to I/O locations 3F7H or 377H, that configuration should be selected in preference to the configuration which also includes 3F7H or 377H.

This method requires that either the socket or the ATA PC Card have the ability to selectively disable port 3F7H or 377H while keeping the other addresses in the Primary or Secondary address range active. This method also requires that the host software shall not attempt to use information in the Drive Address Register.

4. If socket hardware in the system is designed specifically to avoid this conflict then it shall be able to selectively force the socket's system data bus signal D7 to be in high impedance and the PC Card's -IOIS16 signal (-IOCS16 on the host ISA or EISA bus) to treated as negated during accesses to I/O address 3F7H or 377H. This feature would be used when an ATA card is installed. If a floppy disk controller PC Card is permitted to be installed in the system, then each socket must also have the ability to force the socket's system data bus signal lines D6 through D0 to be in high impedance and the card's -IOCS16 signal to treated as negated during accesses to I/O address 3F7H or 377H.

This method requires special socket hardware. This method does not require any special treatment or modifications to existing software accessing the drive at the primary addresses.

APPENDIX - C

CARD INFORMATION STRUCTURE

CARD INFORMATION STRUCTURE

C.1 Card Information Structure

A Card Information Structure shall be present on the card. The minimum required tuples for the card are not necessarily in order of appearance:

a) **Required:** Device ID Tuple, CISTPL_DEVICE, tuple code 01H. This tuple must be the first tuple on the card. If the card supports the memory mapped PC Card ATA mode, a Device ID tuple shall be present which identifies the region of memory space occupied by the ATA registers as having a device type DEH; Function Specific region. In PCMCIA Release 1.0 and 2.0 nomenclature, this device type was named "I/O".

b) **Required:** Configuration Tuple, CISTPL_CONFIG, tuple code 1AH. This tuple identifies the location and presence of the Card Configuration registers in the attribute memory space of the card. In PCMCIA Release 2.0 nomenclature, this tuple was labelled "CISTPL_CONF".

c) **Required:** Configuration Entry Tuples, CISTPL_CFGTABLE_ENTRY, tuple code 1BH. One of these tuples shall be present for each Configuration Index value which is supported by the card. In Release 2.0 nomenclature, this tuple was labelled "CISTPL_CE".

d) **Required:** Card Function ID Tuple, CISTPL_FUNCID, tuple code 21H, with a function ID value of 04H for disk function. This tuple allows Card Services clients to quickly determine the class of card which is present in the socket. See section C.2.

e) **Required:** Disk Function Extension Tuple, CISTPL_FUNCE, tuple code 22, Type 1, Disk Interface with an interface ID value of 01, PC Card ATA interface. This tuple identifies the card as being PC Card ATA. See section C.3.

f) **Recommended:** Disk Function Extension Tuple, CISTPL_FUNCE, tuple code 22, Type 2, PC Card ATA Features. This optional tuple identifies optional features of the PC Card ATA protocol which are implemented on the card. See section C.4.

g) **Recommended:** JEDEC ID Tuple, CISTPL_JEDEC C, tuple code 18H. It is recommended that cards which support the memory mapped ATA registers described in this document identify the region of memory space containing the registers with a JEDEC ID code indicating PC Card ATA protocol support. Use of a standardized ID from section C.5 is recommended, although a vendor specific JEDEC ID codes is permitted. If a vendor specific JEDEC ID is used, the interoperability of the card will be limited to those host systems which recognize vendor's unique IDs.

C.2 Function ID Tuple for Disk Function

This tuple specifies that the card supports disk device functionality. This tuple is followed by one or more Function Extension Tuples which further specify the disk function which is supported. The PC Card ATA CIS shall include this tuple and a Function extension tuple describing the disk interface protocol as PC Card ATA. Additional Function Extension tuples describing the disk function on the card are also permitted.

Offset	CIS	Tuple	Comments	Fields
00H	21H	cistpl_funcid e	Function ID tuple	Tuple Code
02H	02H	link	this tuple has 2 info bytes	Link Length
04H	04H	tplfid_function	Disk Device Function tuple	Function Type
06H	xxH	tplfid_sysinit	System Initialization Flags	Initialization Options

C.3 Disk Device Interface Function Extension Tuple

This tuple specifies the device interface protocol used in the disk function described in the Function ID tuple of section C.2. This tuple shall follow the Function ID tuple for Disk Function described in section C.2 without any other intervening Function ID tuples.

Offset	CIS	Tuple	Comments	Fields
00H	22H	cistpl_funce	(Disk) Function Extension tuple	Tuple Code
02H	02H	link	this tuple has 2 info bytes	Link Length
04H	01H	tplfe_type	Disk Device Interface tuple	Extension Type
06H	01H	tplfe_data	PC Card ATA Interface	Interface Type

C.4 PC Card ATA Features Function Extension Tuple

This function extension tuple specifies the PC Card ATA related features of the disk function described in the Function ID tuple of section C.2. This tuple is optional, but when present, shall follow the Function ID tuple for Disk Function described in section C.2 without any other intervening Function ID tuples.

<table>
<tr><th>Offset</th><th>CIS</th><th>Tuple</th><th>Comments</th><th colspan="8">Bit Fields</th></tr>
<tr><td>00H</td><td>22H</td><td>cistpl_funce</td><td>(ATA Function Extension tuple</td><td colspan="8">Tuple Code</td></tr>
<tr><td>02H</td><td>03H</td><td>link</td><td>this tuple has 3 info bytes</td><td colspan="8">Link Length</td></tr>
<tr><td>04H</td><td>02H</td><td>tplfe_type</td><td>Basic PC Card Extension tuple</td><td colspan="8">PC Card ATA Basic Features</td></tr>
<tr><td>06H</td><td>xxH</td><td>tplfe_data</td><td>PC Card ATA Features Byte 1</td><td>R</td><td>R</td><td>R</td><td>R</td><td>U</td><td>S</td><td colspan="2">V</td></tr>
<tr><td>08H</td><td>xxH</td><td>tplfe_data</td><td>PC Card ATA Features Byte 2</td><td>R</td><td>I</td><td>E</td><td>N</td><td colspan="4">P</td></tr>
</table>

Name	Description	Values
V	Vpp Power	0 Not Required 1 Required for Media Modification Accesses 2 Required for all Media Accesses 3 Required Continuously
S	Silicon	0 Rotating Device 1 Silicon Device
U	Unique Drive Identifier	0 Identify Drive Model / Serial Number may not be unique 1 Identify Drive Model / Serial Number is guaranteed unique
R	Reserved	This field is reserved for future standardization. This bit shall be 0.

P	**Low Power Modes** (Idle, Standby, Sleep)	Bit 3: 0 Low Power Mode Use Required to Minimize Power Bit 3: 1 Drive Automatically Minimizes Power. No need for host to actively power manage. Bit 2: 0 Idle Mode Not Supported Bit 2: 1 Idle Mode Supported Bit 1: 0 Standby Mode Not Supported Bit 1: 1 Standby Mode Supported Bit 0: 0 Sleep Mode Not Supported Bit 0: 1 Sleep Mode Supported
N	3F7/377 Register Inhibit Available	0 All Primary and Secondary I/O Addressing Modes include ports 3F7 or 377. 1 Some Primary or Secondary I/O Addressing Modes exclude 3F7 and / or 377 for floppy interference avoidance.
E	Index Emulated	0 Index Bit is Not Emulated 1 Index Bit is Supported or Emulated
I	IOis16 on Twin Card	0 -IOis16 use is Unspecified on Twin-Card Configurations 1 -IOis16 is asserted only for Data Register on Twin-Card Configurations
R	Reserved	This field is reserved for future standardization. This bit shall be o.

C.5 PC Card ATA JEDEC ID's

The *PCMCIA PC Card Specification*, Release 2.01 and previous releases, provide for the use of JEDEC Identifiers to specify the access algorithm for regions of memory space on a PC Card. PCMCIA has adopted specific JEDEC ID codes to indicate regions of memory space which contain the memory mapped PC Card ATA registers as described in this document. These identifiers use the JEDEC Manufacturer ID of 95 decimal of DFH, which has been assigned to PCMCIA by JEDEC. The following two byte JEDEC ID's are used for PC Card ATA regions. The JEDEC ID's for PC Card ATA identify both the access protocol, PC Card ATA, and the handling of the Vpp supply within the PC Card ATA protocol.

The JEDEC ID's for PC Card ATA are defined as follows:

First Byte	Second Byte	Description
DFH	01H	PC Card ATA with no Vpp required for any operation.
DFH	02H	PC Card ATA with Vpp required for media modification operations only.
DFH	04H	PC Card ATA with Vpp required for all media access.
DFH	08H	PC Card ATA with Vpp required continuously.

OVERVIEW

This section specifies an AIMS (Auto Indexing Mass Storage) Card standard for storing large data sets such as image and multimedia data files. It describes a standard card interface for an electronic camera or other portable devices, and also accessible by a variety of computer platforms. The protocol uses the existing *PCMCIA PC Card Standard,* Release 2.01 defined pinout without any changes or additional pins. Additionally, the protocol is backward compatible and can be accessed with PCMCIA Release 1.0 sockets.

The card is architected as a block oriented mass storage device with a view to implementation with Flash EEPROM technology. Access is through a set of registers which contain the parameters necessary to read, write and erase data from the card. The card can be configured to operate as either an I/O or a memory device.

1.1 Advantages

1. Interface is accessible to consumer electronics type hosts. Only requires support of an 8-bit data bus, 7 address lines and control signals.
2. Access is in either memory mapped (polled program control) or I/O mapped (interrupt driven control) operation.
3. Memory technology is isolated - various memory technologies all have a common interface. This is particularly important for Flash EEPROM's.
4. Supports an erase ahead mode to enable a fast write with Flash EEPROM.
5. Completely Release 1.0, 2.0 and 2.01 compatible.

1.2 Requirements

1. Requires a separate device driver for this card function.

SECTION - 2

PIN ASSIGNMENTS

PIN ASSIGNMENTS

2.1 Pin Assignments

Table 2-1: AIMS Card Pin Assignments

Pin	Signal	I/O	Function
1	GND	I	Ground
2	D3	I/O	Data bit 3
3	D4	I/O	Data bit 4
4	D5	I/O	Data bit 5
5	D6	I/O	Data bit 6
6	D7	I/O	Data bit 7
7	-CE1	I	Card Enable
8	NC		No Connect
9	-OE	I	Output Enable
10	NC		No Connect
11	NC		No Connect
12	NC		No Connect
13	NC		No Connect
14	NC		No Connect
15	-WE/-PGM	I	Write Enable/Program
16	RDY/-BSY* (-IREQ)	O	Ready/Busy (Interrupt Request)
17	Vcc	I	+5 volts
18	Vpp1	I	Program voltage
19	NC		No Connect
20	NC		No Connect
21	NC		No Connect
22	A7	I	Address bit 7
23	A6	I	Address bit 6
24	A5	I	Address bit 5
25	A4	I	Address bit 4
26	A3	I	Address bit 3
27	A2	I	Address bit 2
28	A1	I	Address bit 1
29	NC		No Connect
30	D0	I/O	Data bit 0
31	D1	I/O	Data bit 1
32	D2	I/O	Data bit 2
33	NC		No Connect
34	GND		Ground
35	GND		Ground
36	-CD1	O	Card Detect
37	NC		No Connect
38	NC		No Connect
39	NC		No Connect
40	NC		No Connect
41	NC		No Connect
42	NC		No Connect
43	NC		No Connect
44	-IORD	I	I/O Read
45	-IOWR	I	I/O Write
46	NC		No Connect
47	NC		No Connect
48	NC		No Connect
49	NC		No Connect
50	NC		No Connect
51	NC		No Connect
52	NC		No Connect
53	NC		No Connect
54	NC		No Connect
55	NC		No Connect
56	NC		No Connect
57	NC		No Connect
58	RESET	I	Card Reset
59	-WAIT	O	Bus Cycle Wait
60	-INPACK	O	Input Port Acknowledge
61	-REG	I	Register Select
62	NC		No Connect
63	-STSCHG	O	Card Status Change
64	NC		No Connect
65	NC		No Connect
66	NC		No Connect
67	-CD2	O	Card Detect
68	GND		Ground

Notes:

1. I - Input to card, O - Output from card, I/O - Bidirectional.

*2. Pin 16 becomes IREQ in I/O mode.

3. Pins indicated as NC may actually be connected in accordance with the*PCMCIA PC Card Standard*, Release 2.01 either on the card side or the host side or both.

SECTION - 3

REGISTER SET

REGISTER SET

3.1 Register Set Definition

The main memory on the AIMS Card is accessed through the card registers. The AIMS Card defines five registers, in addition to the two required by *PCMCIA PC Card Standard,* Release 2.01 (Pin Replacement Register and Configuration Option Register). They are:

Address Register set	4 bytes	(32-bits)	contains REG0, REG1, REG2, REG3
Data Register	1 byte	(8-bits)	
Command Register	1 byte	(8-bits)	
Mode Register	1 byte	(8-bits)	
Block Count Register set	2 bytes	(16-bits)	contains COUNT LOW, COUNT HIGH

The host accesses the card registers through the following control modes,

AIMS Card specific	PCMCIA PC Card Standard Release 2.01
Read/Write Address	Read/Write Configuration Option
Read/Write Data	Read/Write Pin Replacement
Read/Write Mode	Read Attribute Memory (does not use registers)
Write Command	
Write Block Count	

The general procedure for executing a card function is to set up the Address Register, load the Command Register, check the Ready/Busy state for completion, and check the Mode Register for status.

To the host, the registers on the card may reside in either memory space or I/O space. Memory mapped modes use the Output Enable (OE-) and Write Enable (WE-) lines, while I/O mapped modes use the I/O Read (IORD-) and I/O Write (IOWR-) lines. It is the responsibility of the host to assign the registers in the host space; for a PC host, it is Socket Services that assigns the registers to the host space.

Note that the same Ready/Busy function appears in three places. These are the RDY/BSY pin, a bit in the Mode Register, and a bit in the Pin Replacement Register. The information is provided redundantly for ease of use under different host access protocols.

The PCMCIA standard combines mapping mode and access protocol, so that memory mapped mode uses the Ready/Busy access protocol, and I/O mapped mode uses interrupt requests. In memory mapped mode, the Ready/Busy function is available on the RDY/BSY pin and in the Mode Register, while in I/O mode the Ready/Busy function is available in the Pin Replacement Register and the Mode Register.

The Pin Replacement and Configuration Option Registers are used in the I/O mode and operate as described in the *PCMCIA PC Card Standard* , Release 2.01, Section 4.15. Since all control functions may be accessed with the card registers in memory mapped mode, the AIMS Card can be read by a Release 1.0 socket.

It is recommended that the AIMS Card not rely on the WAIT signal, for compatibility with a *PCMCIA PC Card Standard* , Release 1.0 socket. A Release 1.0 socket may not be able to access the card at normal card timing, so extended delay timing or special protocols may be required.

3.2 Register Set Functions

3.2.1 Addressing the Card Memory (Read/Write Address Register)

The memory of the card is addressed by the 32-bit Address Register set on the card. This register is loaded through the data bus 8-bits at a time. The full address points to the data byte available on the data bus when Read Data or Write Data is initiated.

The Address Register set automatically increments in conjunction with each main memory read or write operation. This Register is valid following an EORD or EOWR command, but not necessarily valid during operations when it does internal sequencing. The auto increment feature allows a block of data to be transferred without rewriting the Address Register.

On Reset, it is recommended that the Address Register Set be initialized to zero.

3.2.2 Main Memory Read Function (Read Data Register)

This register is used to transfer data bytes from the card to the host. The data byte addressed by the Address Register set is then available to the host after one access time. The card may assert the Wait or Ready/Busy lines as needed for access time considerations. Additional data bytes in sequence are available with each additional Read Data Register cycle without rewriting the Address Register set, because of the autoincrement feature. The size of a read block is specified in the CIS format tuple in TPLFMT_RBSZ.

3.2.3 Main Memory Write Function (Write Data Register)

This register is used to transfer data bytes from the host to the card. The data byte is written to the card at the address specified by the contents of the Address Register set. The card may assert the Wait and Ready/Busy lines as needed for access time considerations, and during this time the signals to the card must be held constant. Additional data bytes will be written in sequence with each additional Write Data Register cycle without rewriting the Address Register set, because of the auto increment feature. The size of a write block is specified in the CIS format tuple in TPLFMT_WBSZ.

3.2.4 Read Attribute Memory Function

Attribute memory is read directly through lines A1 - A7, without using registers. The number of address lines limits the amount of attribute memory that can be read to 128 bytes.

3.2.5 Erase Function (Write Block Count Register)

The main memory is erased by loading the Address Register with the starting address of the memory to be erased, loading the Block Count Register set (Low and High) with the number of consecutive blocks to be erased, and then issuing an erase command to the Command Register. The size of an erase block is specified in the CIS format tuple in TPLFMT_EBSZ.

3.2.6 Status and Control Functions (Mode Register)

The mode register gives the status of card functions and controls operation of the card control circuitry. The definitions of the 8-bits of the Mode Register are shown in Table 3-1.

Error bits are cleared at the beginning of each command and are valid at the end of each command. When cleared, the contents of EC0 - EC3 are undefined.

Table 3-1: Mode Register Bit Definitions

D7	D6	D5	D4	D3	D2	D1	D0
RDY/BSY	IRQ ACK	IEN	ERR DET	EC3	EC2	EC1	EC0

ECn Error codes 0-3. Read only bits. Vendor specific.

ERR DET Error detect. Read only bit.
Error = 1, no error (successful operation) = 0.

IEN Interrupt enable. Read/write bit.
Enable =1, no interrupt enable = 0.

IRQ ACK Interrupt request acknowledge. Read/write bit.
On read, = 1 for interrupt request pending,
On write, =1 for interrupt request acknowledge (IRQ ACK)

RDY/BSY Ready/Busy. Same as Pin Configuration Register.
Read only bit. Ready = 1, Busy = 0.
BSY (Busy) is cleared whenever the card has access to the Command Block Registers. The host should not access the Command Block Registers when Busy=0.

3.2.7 Error Handling

Five bits in the mode register are used to signal error conditions. Bit D4 is used as an error detect bit for software routines (1=set, error has occurred; 0= clear, no error). The remaining four bits, D3-D0, are used to designate types of error conditions, as shown in Table 3-2. A given error condition may cause multiple bits to be set.

All fatal errors will issue an interrupt to the host, if in I/O mode.

Table 3-2: Error Bit Definitions

Bit Number	Error Type
D0	Data Error
D1	Command Error
D2	Access Error
D3	Controller Error

Table 3-3: Error Codes

Error Code	Description
00H	No error
01H	Read error
02H	Write error
03H	Erase error
04H	Write verify error
05H	Erase verify error
06H	Verify error
07H	Uncorrectable data error
08H	Corrected data
09H	Bad data
0AH	Invalid address
0BH	No program voltage
0CH	Invalid command
0DH	Timeout
0EH	End of address range
0FH	Write protect error
10H - 7FH	Reserved for future use (RFU)
80H - FFH	Vendor unique codes

3.3 Control Modes

Control modes for the two registers required for compatibility with *PCMCIA PC Card Standard*, Release 2.01, the Configuration Option Register and the Pin Replacement Register, are described in Table 3-4. These two registers are used in I/O mode but reside in attribute memory; therefore this table does not describe any I/O mapping. The Read Attribute Memory value in line 5 means the value at the address location specified by address lines A1 through A7, which will be greater than or equal to zero, and less than TPCC_RADR. Note that the address offset TPCC_RADR is fixed, at a value of E0 hex, for the purpose of saving space in the CIS.

The control modes for the registers specific to the AIMS Card are given in Table 3-5. The read portion of write-only registers (e.g. Command, Count) and write portion of read-only registers are reserved.

Table 3-4: PCMCIA I/O Card Register Control Modes, Attribute Memory

Function Mode	-REG	-CE	-OE	-WE	Address Offset from TPCC_RADR
Write Configuration Option Register	L	L	H	L	00H
Read Configuration Option Register	L	L	L	H	00H
Write Pin Replacement Register	L	L	H	L	04H
Read Pin Replacement Register	L	L	L	H	04H
Read Attribute Memory	L	L	L	H	lines A1-A7

Table 3-5: AIMS Card Register Control Modes, Common Memory (I/O) Mapped

Function Mode	-REG	-CE	-OE (-IORD)	-WE (-IOWR)	Address Offset
Write Address REG 0	H(L)	L	H	L	00H
Write Address REG 1	H(L)	L	H	L	02H
Write Address REG 2	H(L)	L	H	L	04H
Write Address REG 3	H(L)	L	H	L	06H
Read Address REG 0	H(L)	L	L	H	00H
Read Address REG 1	H(L)	L	L	H	02H
Read Address REG 2	H(L)	L	L	H	04H
Read Address REG 3	H(L)	L	L	H	06H
Write Block Count Low	H(L)	L	H	L	08H
Write Block Count High	H(L)	L	H	L	0AH
Write Command Register	H(L)	L	H	L	0CH
Write Mode Register	H(L)	L	H	L	0EH
Read Mode Register	H(L)	L	L	H	0EH
Write Data Register	H(L)	L	H	L	10H
Read Data Register	H(L)	L	L	H	10H
Vendor Unique Register	H(L)	L	-	-	12H
Vendor Unique Register	H(L)	L	-	-	14H
Vendor Unique Register	H(L)	L	-	-	16H
Reserved - all other Registers	H(L)	L	-	-	18H .. 1EH

SECTION - 4

COMMAND SET

COMMAND SET

4.1 Command Codes

Table 4-1: Command Codes

Type	Command	Code
D	Write Blocks (without erase)	38_H
D	Read Blocks	40_H
C	Erase Blocks	$C0_H$
C	Execute Diagnostic	90_H
C	End of Write (EOWR)	$4F_H$
C	End of Read (EORD)	$3F_H$
C	Enter Diagnostic Mode	88_H
C	Return Internal Error Code	$9A_H$
D	Write Verify Blocks (optional)	$3C_H$
	Vendor Unique	$F0_H$ - FF_H

Note 1: The block size is defined in the CIS for read, write and erase. Each case is 2n bytes.

Note 2: Type C runs to a known completion point.
Type D runs until Stop command (e.g., EOWR, EORD) is issued or error occur.

Note 3: The "Erase Blocks" command means the block is prepared for writing.
This standard does not specify the state of the contents of the block after execution of the Erase Blocks command.

Note 4: The description of command codes includes the timing diagrams and logic protocols in Figures 5-1 through 5-6.

Note 5: All command codes not specifically designated above are reserved for future use.

4.2 Overview of Command Set Definition

This section defines the commands used to control the AIMS Card operation. The command codes are given in Table 4-1. The commands are based on those used for hard disk storage but are adapted for solid state storage, particularly for Flash EEPROM. One of the added command set features is the Erase command, which allows a given segment of memory to be pre-erased and then written to at a later time. This 'erase ahead' feature substantially increases system write speed for implementations using Flash EEPROM.

There are two types of commands, designated "C" and "D". Type "C" commands run to a defined end state. These commands execute a specific action such as erasing a number of blocks. Type "D" commands execute repetitively until stopped or error occur. For example, when the Read Blocks command is loaded into the Command Register, a byte is placed on the data bus with every access. These bytes are read from sequential addresses starting at the base address loaded into the Address Register.

All memory addresses must start on block boundaries or else the card will generate an error status in the Mode Register. The only legal command accepted after a Read/Write Type "D" command is the corresponding EORD/EOWR command. Read and Write commands are mutually exclusive and can be used only sequentially, not both at the same time. After each block is transferred or after each command is complete, the status register may be checked for error conditions.

Following is a detailed description of each command, its effect, and the registers used.

4.3 Command Set Definition

4.3.1 Write Blocks

Registers Used: Address 0-3, Command, Status, Data Type=D Code=38H

Purpose: Write a number of blocks.

This command assumes that the area to be written to has been "pre-erased", if required by the particular memory technology being implemented. This allows the card to be operated in the 'erase ahead' mode discussed above.

Before issuing this command, the first address to be written to is loaded into the Address Registers 0-3. Then the Write Blocks code (38H) is loaded into the Command Register (Address 0CH). On execution of this command, each byte written to the Data Register (Address 10H) will be written to the on-card memory at sequential addresses, starting at the base address in Address Registers 0-3. It is the responsibility of the host to address on whole block boundaries and to write only to 'pre-erased' blocks if that is appropriate.

On write operation, if no error occurs, the Address Register Set points to the next available block to be written. However, if error occurs, the Address Register Set will point to the error block.

This command is not allowed to wrap addresses, from high values back to zero, as the address is automatically incremented. However, because this is a low level command, the card is capable of writing past the end of a partition and into the next partition (as defined by a Layer 2 format tuple in the CIS).

4.3.2 End of Write (EOWR)

Registers Used: Command, Status Type=C Code=4FH

Purpose: Terminate the Write Blocks or Write Verify Blocks command.

When in the Write Blocks mode, the card continues to write data to the on-board memory each time the Data Register is written to. To terminate the Write Blocks mode, an EOWR command code (4FH) is written to the Command Register (Address 0CH). On receiving this command, the card should write any remaining data from the card buffers to the current block and terminate the write operation. Partial blocks may be filled with padded data, but the value of such padded data is not specified by this standard.

It is the responsibility of the card to notify the host of completion of the EOWR cycle by either 1) asserting the RDY/BSY line if in memory mode, 2) asserting the IREQ line if in I/O mode or 3) setting the appropriate bit in the Mode Register, in either mode. Error conditions are reported by setting the appropriate bits in the Mode Register.

4.3.3 Read Blocks

Registers Used: Address 0-3, Command, Status, Data Type=D Code=40H

Purpose: Read a number of blocks.

Before issuing this command, the first address to be read from is loaded into the Address Registers 0-3. Then the Read Blocks code (40H) is loaded into the Command Register (Address 0CH). On execution of this command, each byte read from the Data Register (Address 10H) will be accessed from the on-card memory at sequential addresses, starting at the base address in Address Registers 0-3. It is the responsibility of the host to start reading on block boundaries.

On read operation, if no error occurs, the Address Register Set points to the next available block to be read. However, if error occurs, the Address Register Set will point to the error block.

This command is not allowed to wrap addresses, from high values back to zero, as the address is automatically incremented. However, because this is a low level command, the card is capable of writing past the end of a partition and into the next partition (as defined by a Layer 2 format tuple in the CIS).

4.3.4 End of Read (EORD)

Registers Used: Command, Status Type=C Code=3FH

Purpose: Terminate the Read Blocks command.

When in the Read Blocks mode, the card continues to read data from on-board memory each time the data register is read from. To terminate the Read Blocks mode, an EORD command code (3FH) is written to the Command Register (Address 0CH). On execution of the command, the card will terminate the read operation.

It is the responsibility of the card to notify the host of completion of the EORD cycle by either 1) asserting the RDY/BSY line if in memory mode, 2) asserting the IREQ line if in I/O mode or 3) setting the appropriate bit in the Mode Register. Error conditions should be reported by setting the appropriate bits in the Mode Register.

4.3.5 Erase Blocks

Registers Used: Address 0-3, Block Count High, Block Count Low, Command, Status
Type=C Code=C0H

Purpose: Erase a number of blocks.

Before issuing this command, the first address to be erased is loaded into the Address Registers 0-3. It is the responsibility of the host system to ensure this occurs on a block boundary. The number of sequential blocks to be erased is then loaded into the registers Block Count Low (Address 08H) and Block Count High (Address 0AH). Then the Erase Blocks code (C0H) is loaded into the Command Register (Address 0CH). On execution of this command, the number of blocks of memory specified in the Block Count Low/High Registers will be placed in an "erased" state, starting at the base address in Address Registers 0-3. An erased state is defined to mean that the memory block are now ready to be written to without further processing.

On erase operation, if no error occurs, the Address Register Set points to the next available block to be erased. However, if error occurs, the Address Register Set will point to the error block.

It is the responsibility of the card to notify the host of completion of the erase cycle by either 1) asserting the RDY/BSY line if in memory mode, 2) asserting the IREQ line if in I/O mode or 3) setting the appropriate bit in the Mode Register. Error conditions should be reported by setting the appropriate bits in the Mode Register.

4.3.6 Execute Diagnostic

Registers Used: Command, Status Type=C Code=90H

Purpose: Execute a vendor specific diagnostic routine for the card.

The Execute Diagnostic code (90H) is written to the Command Register (Address 0CH), to begin routine diagnostics of the card. It is the responsibility of the card to notify the host of completion of the diagnostic cycle by either 1) asserting the RDY/BSY line if in memory mode, 2) asserting the IREQ line if in I/O mode or 3) setting the appropriate bit in the Mode Register. Error conditions should be reported by setting the appropriate bits in the Mode Register.

SECTION - 5

CONTROL MODE PROTOCOLS

CONTROL MODE PROTOCOLS

Figures 5-1 through 5-3 show the timing diagrams for read, write and erase functions. Figures 5-4 through 5-6 show the corresponding logical flow protocols for the read, write and erase functions.

The assumptions for these card control protocols are,

1. The minimum read, write, and erase block sizes are defined by the CIS.
2. Before writing begins, a block is assumed to be ready to be written to, (i.e., pre-erased for EEPROM). The purpose of separating the erase and write functions is to speed write operations for EEPROM devices.
3. Data within the card is allowed to be fragmented, so that data can be written to a non-contiguous file. It is the responsibility of the file system (CIS Level 3) to determine how fragmentation will be handled.
4. Host should check Ready/Busy line (or status of BUSY in the Mode Register) after every command line and after every block transfer, in order to provide for exception handling.
5. The maximum address offset used here is 10H, out of a possible available I/O address space size of 3FH. The base address of the AIMS Card in the host I/O space is determined by the host.

Figure 5-1. Read Command Timing Diagram

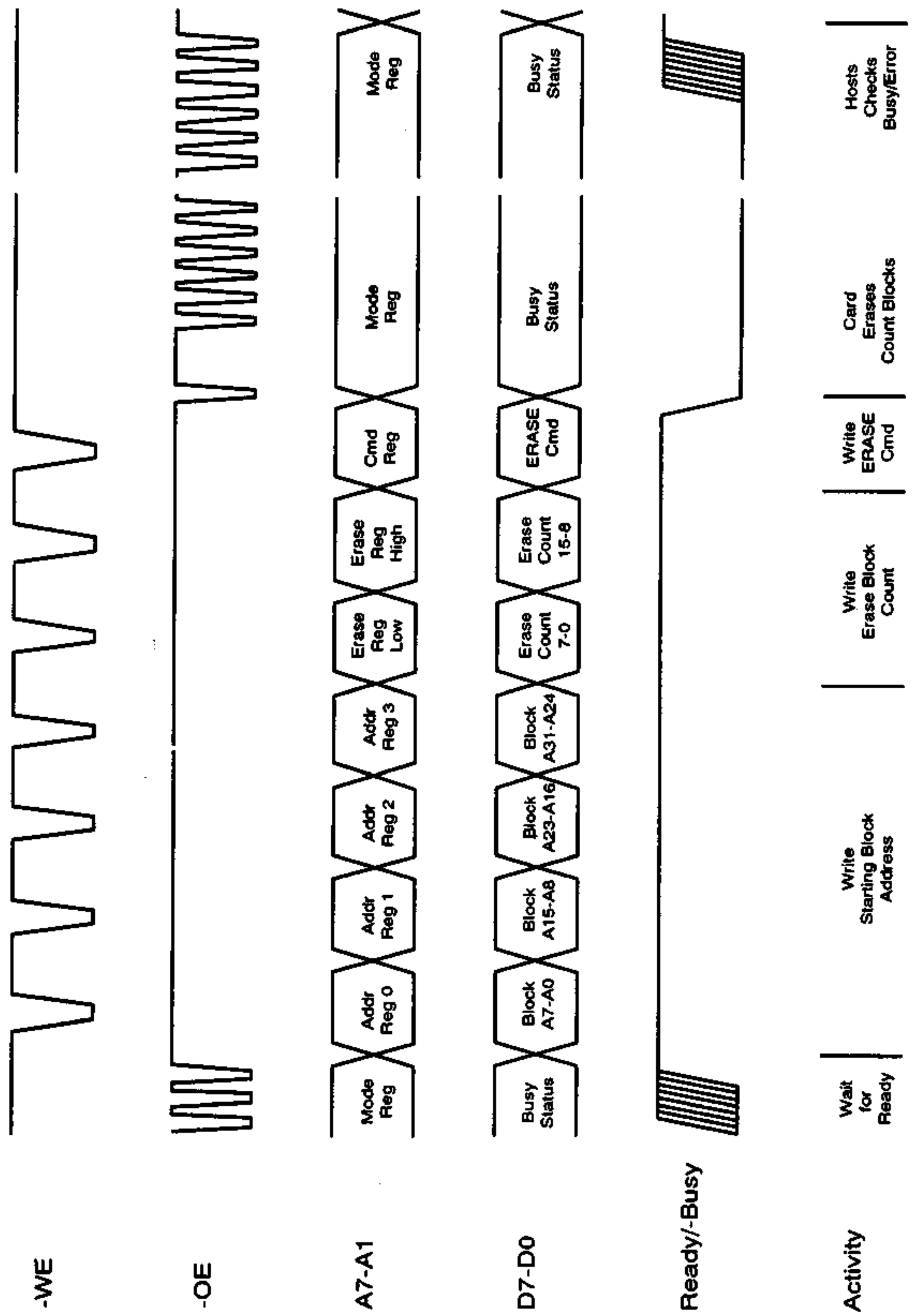


Figure 5-2. Write Command Timing Diagram

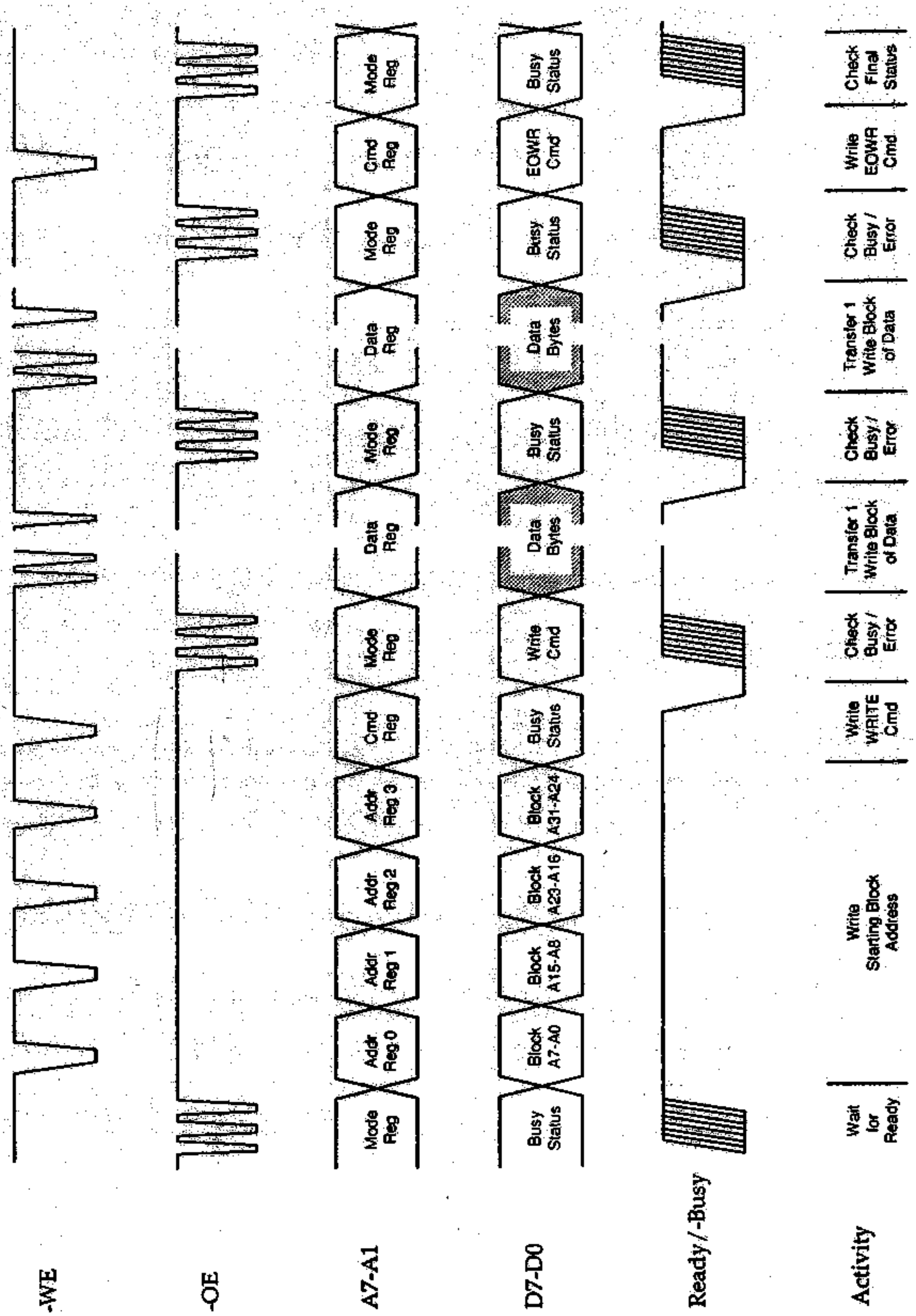


Figure 5-3. Erase Command Timing Diagram

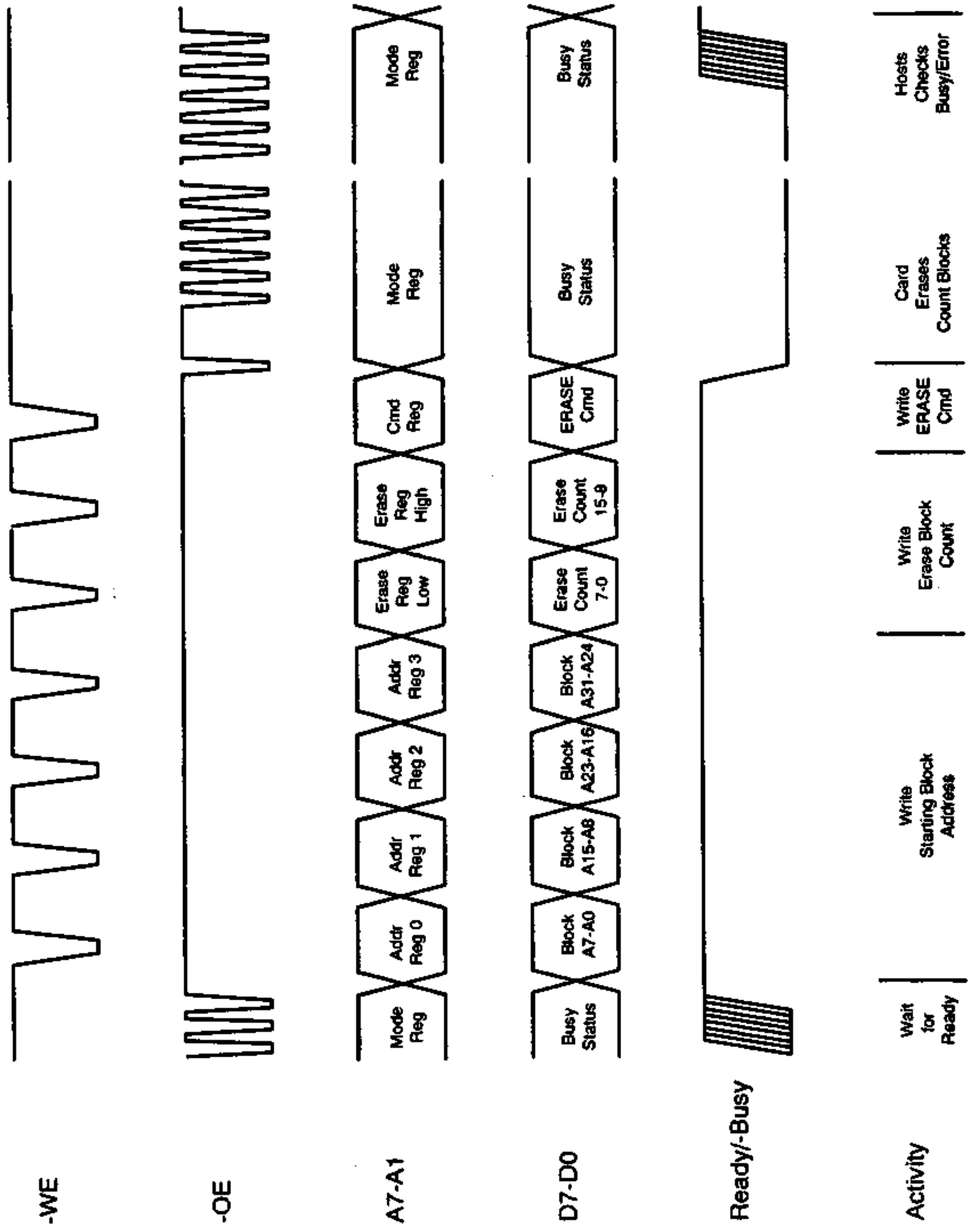


Figure 5-4. Read Function Logic Protocol

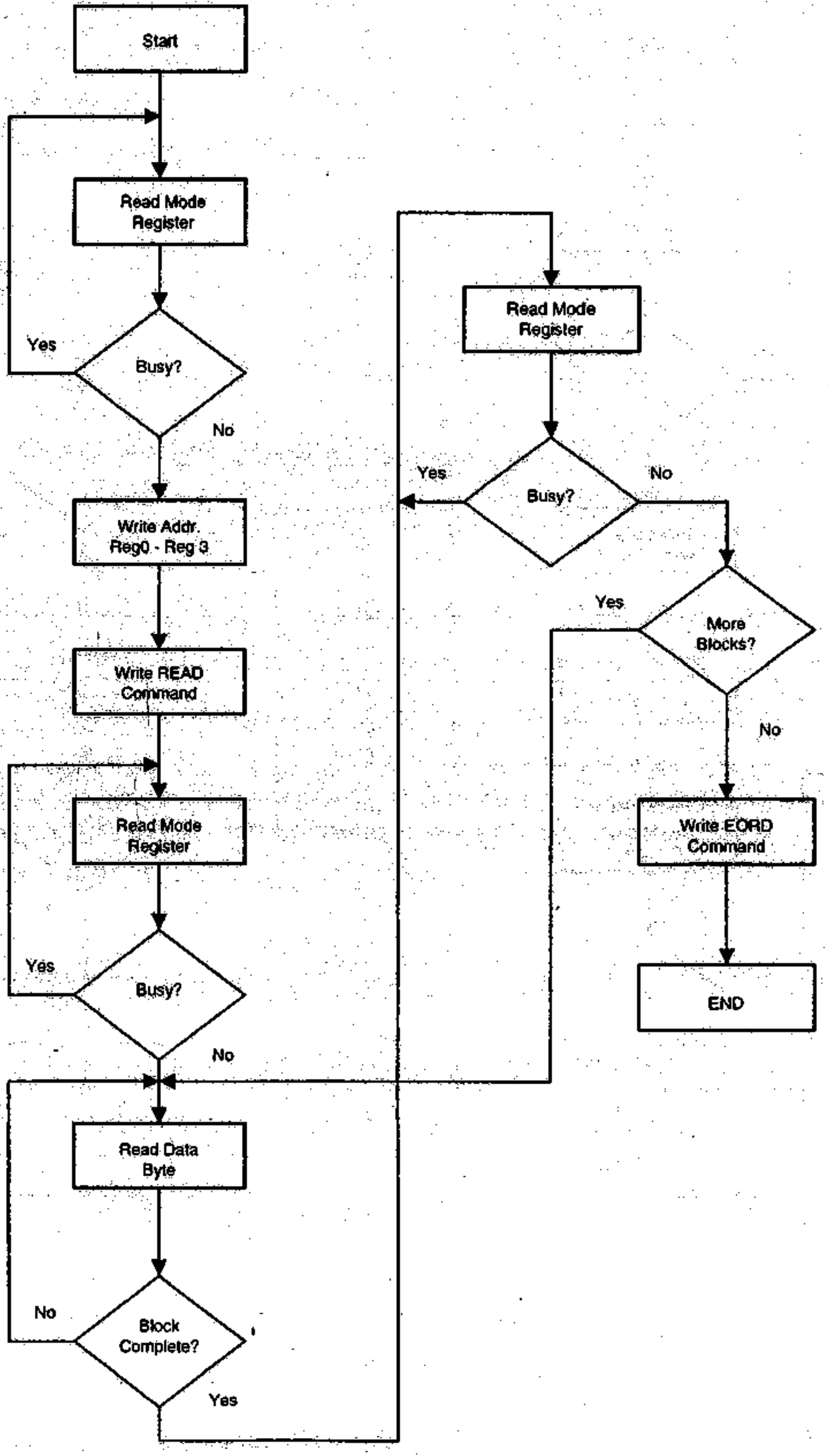


Figure 5-5. Write Function Logic Protocol

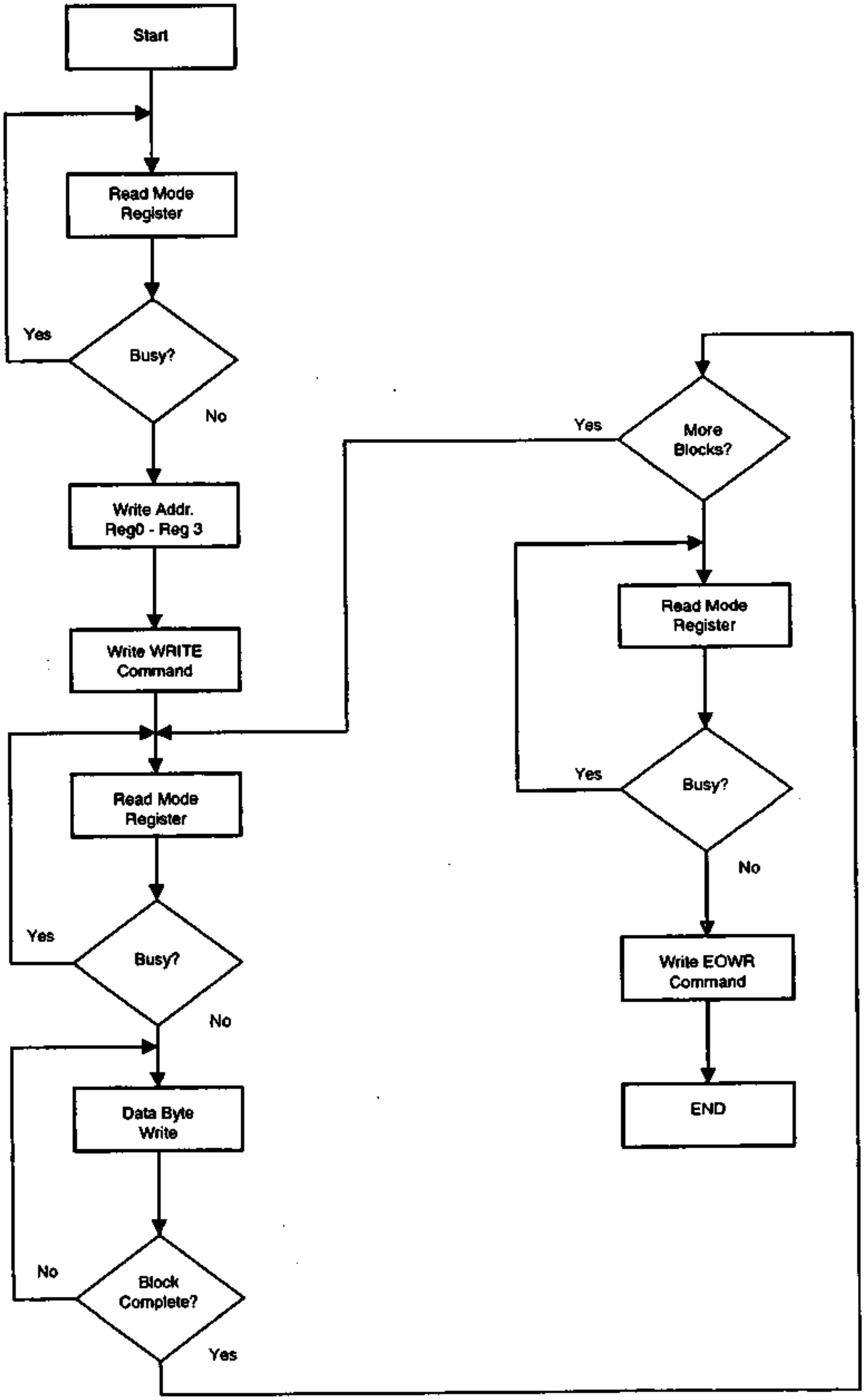
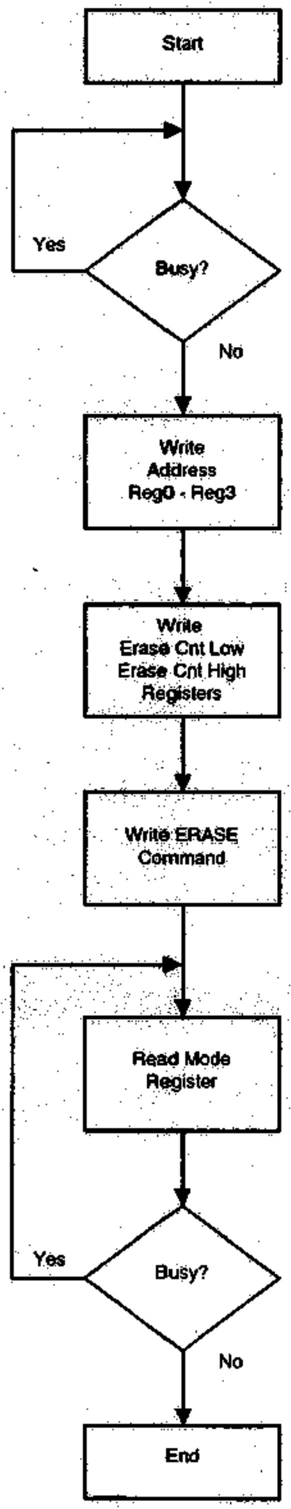


Figure 5-6. Erase Function Logic Protocol


Start
Yes
Busy?
No
Write Address Reg0 - Reg3
Write Erase Cnt Low Erase Cnt High Registers
Write ERASE Command
Read Mode Register
Yes
Busy?
No
End

SECTION - 1

RECOMMENDED EXTENSIONS

RECOMMENDED EXTENSIONS

1.1 Card Physical Package Outline Additions

The Card Physical Committee and the Board of Directors has adopted three new Card Physical PC Card outlines. The three new outlines are:

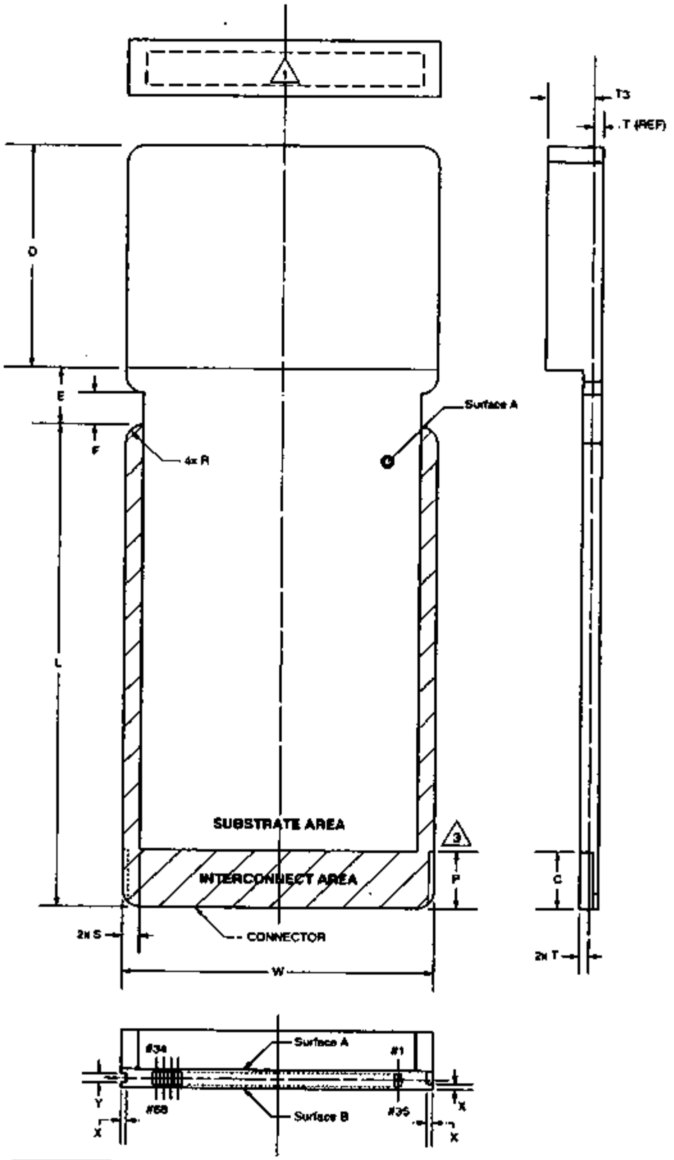
- Type I Extended
- Type II Extended
- Type III

These three outlines exhibit some unique features designed to assist the PC Card designer in the design and implementation of PC Cards with added features.

The Type I and II Extended PC Card outlines were primarily specified so that Designers and Manufacturers of I/O PC Cards may encase their electrical and magnetic isolation devices within the shielded PC Card enclosure. It should be noted that the Type I and II Extended PC Card outlines are identical to the Type I and II PC Card outlines except for the extended portion. The extended portion extends 10 mm past the standard PC Card length of 85.6 mm before the height may be increased in the bubble area. It should be noted that in the extended bubble portion the thickness from the centerline of the connector to the bottom of the PC Card is the same thickness for the entire length. The centerline of the connector to the top of the bubble is 8.0 mm recommended. This will allow a thicker bubble but the designer and manufacturers must consider the ramifications of a thicker than normal bubble (will the card bubble interfere with other devices or covers and will the end user have usage problems). It was noted during the discussion before passage of the Type I and II Extended PC Card Outline that some connectors (RJ-11 and RJ-45) will not fit within the recommended thickness. Therefore the Card Physical Committee specified the thickness such that the centerline of the connector to the bottom of the PC Card is 2.5 mm max. The height from the socket centerline to the top of the bubble is 8.0 mm recommended. This recommended height will allow the designer to increase the total thickness of the bubble in order to accommodate the RJ-11 and RJ-45 connectors.

The Type III PC Card was specified by the Card Physical Committee in order to accommodate thicker devices. It was noted by the Committee that some of the uses for PC Cards may require a body thickness greater than 5.0 mm. Some of the uses discussed were LAN, Fax/Modem, Specialty I/O and Small Form Factor Hard Drives. It should be noted that the original discussions for the Type III PC Card did not include the Small Form Factor Hard Drives. The Type III PC Card Outline appears to be popular for numerous PC Card applications. The Type III PC Card Outline also specifies the PC Card thickness from the centerline to the bottom which is identical to the Type II PC Card (2.5 mm max). The centerline of the connector to the bottom of all P C Card outlines is 2.5 mm max in the substrate area. All PC Card outlines specify 1.65 mm from the centerline to the top or bottom of the PC Card in the interconnect area. By specifying all PC Cards from the centerline of the connector to the bottom and in the interconnect area the same it allows the same Host Connector to be used for all PC Cards specified by PCMCIA. It should be noted that the Type III PC Card outline specifies the thickness from the connector to the top of the substrate area to be 8.0 mm +0/-0.5 mm. The thickness of the substrate area of the Type III PC Card is the thickness of two Type II PC Cards with 0.5 mm clearance between the two PC Cards. It should also be noted that the bubble area on a Type III PC Card is 51 mm max wide and the Type II PC Card bubble are is 48 mm max. When specifying the Host Connector for the PC Cards it is very important to specify to the connector vendor whether a Type III PC Card will be inserted. With the Type III PC Card substrate area width specified at 51 mm max, the guide rails on the Host Connector must be 51 mm wide. It should be noted also that a Host Connector designed to accommodate a Type III PC Card will also accommodate a Type I and II PC Card. A potential misalignment of the PC Card during initial insertion is present, but this Card connector area may allow this misalignment, but the polarization keys on the PC Card are 10 mm long. Once the PC Card is inserted into the Host Connector farther than the polarization key length then this potential misalignment problem will no longer exist.

Figure 1-1. TYPE I Extended PC Card Package Dimensions



C MIN	L ± 0.008	P MIN △3	T △4	S MIN	W ± 0.004	X ± 0.002
0.394 (10.0)	3.370 (85.60)	0.394 (10.0)	0.065 (1.65)	0.118 (3.0)	2.126 (54.0)	.039 (1.00)
Y ± .002	D MAX	E ± .008	F MIN	R MAX	T3 5	
0.063 (1.60)	1.575 (40.0)	0.394 (10.0)	0.197 (5.0)	0.118 (3.0)	0.315 (8.0)	

△1 RECOMMENDED I/O CONNECTORLOCATION.

2. THE PC CARD SHALL BE OPAQUE (NON SEE THRU)

△3 POLARIZATION KEY LENGTH.

△4 INTERCONNECT AREA TOLERANCE = ±.002
SUBSTRATE AREA TOLERANCE = ±.004

△5 RECOMMENDED MAX DIMENSION

6 MILLIMETERS ARE IN PARENTHESIS ( )

Figure 1-2. TYPE I Extended 3-D

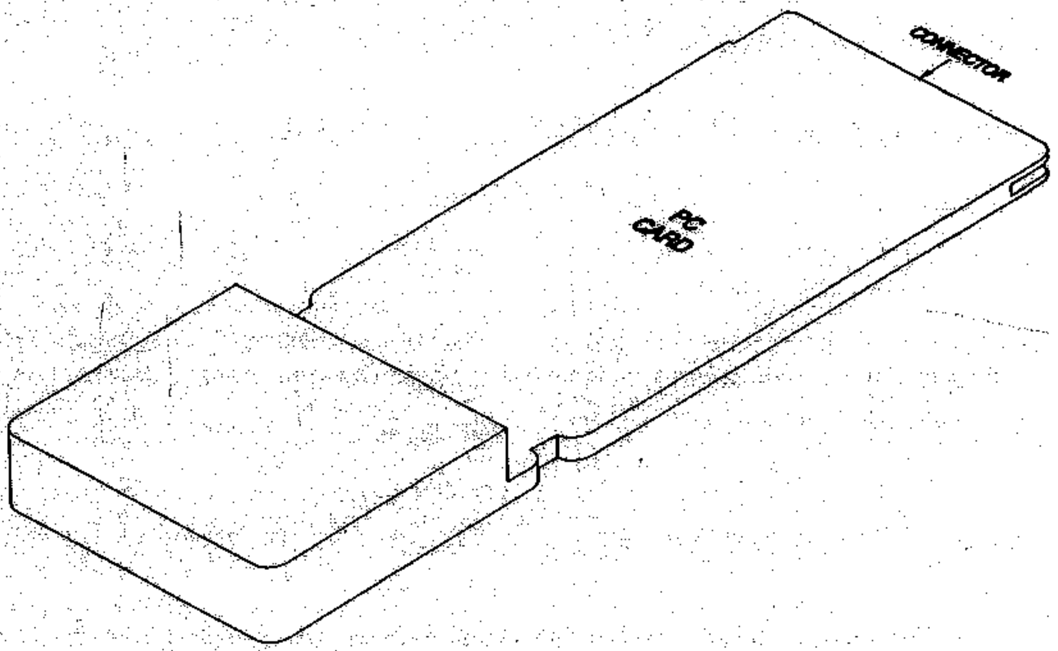
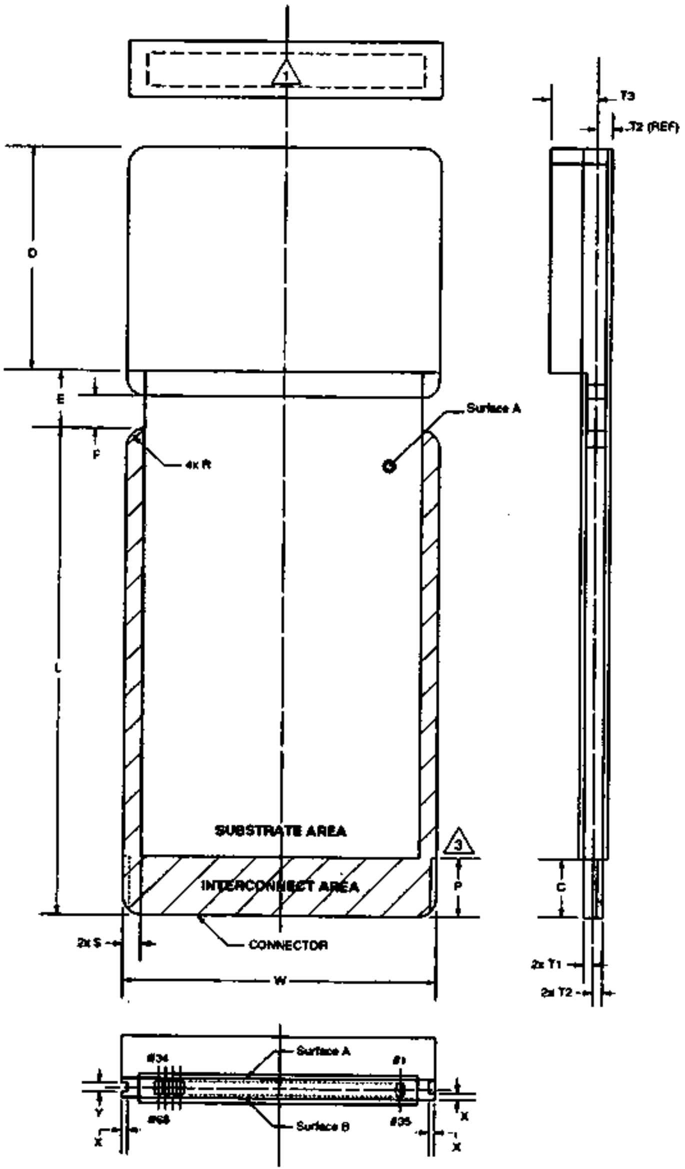


Figure 1-3. TYPE II Extended PC Card Package Dimensions



C MIN	L ± 0.008	P MIN △	T ± 0.002	T2 MAX	S MIN	W ± 0.004
0.394 (10.0)	3.370 (85.60)	0.394 (10.0)	0.065 (1.65)	.098 (2.50)	0.118 (3.0)	2.126 (54.0)
X ± .002	Y ± .002	D MAX	E ± .008	F MIN	R MAX	T3 △
0.039 (1.00)	0.063 (1.60)	1.575 (40.0)	0.394 (10.0)	0.197 (5.0)	0.118 (3.0)	0.315 (8.0)

△1. RECOMMENDED I/O CONNECTOR LOCATION.

2. THE PC CARD SHALL BE OPAQUE (NON SEE THRU)

△3. POLARIZATION KEY LENGTH.

△4. INTERCONNECT AREA TOLERANCE = ±.002
SUBSTRATE AREA TOLERANCE = ±.004

5. MILLIMETERS ARE IN PARENTHESIS ( ).

Figure 1-4. TYPE II Extended 3-D

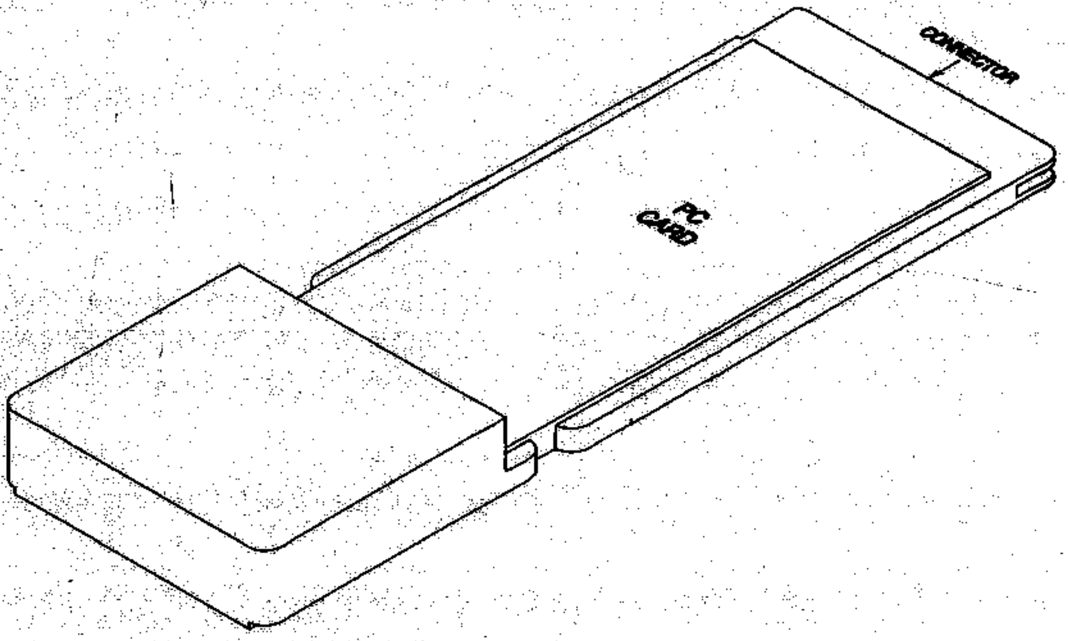
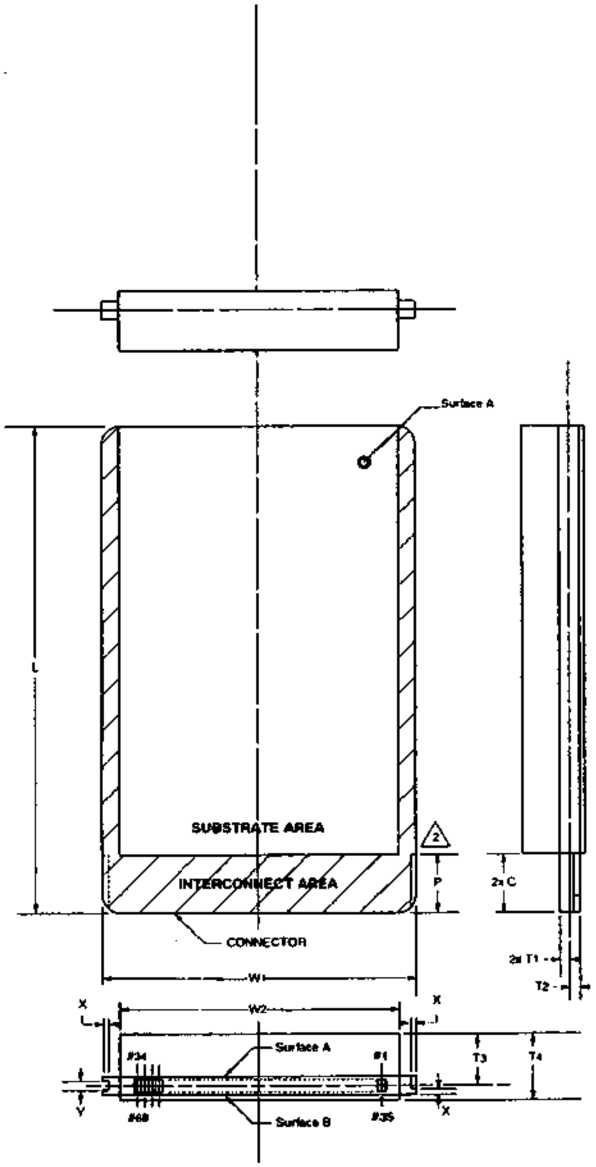


Figure 1-5. TYPE III PC Card Package Dimensions



C MIN	L ± 0.008	P MIN 2	T1 ± 0.002	T2 MAX	T3 $^{+0.008}_{-0.020}$	T4 REF
0.394 (10.0)	3.370 (85.6)	0.394 (10.0)	0.065 (1.65)	0.098 (2.50)	0.315 (8.00)	0.413 (10.50)
W1 ± .004	W2 MAX	X ± 0.002	Y ± 0.002			
2.126 (54.0)	2.008 (51.0)	0.039 (1.00)	0.063 (1.60)			

1. THE PC CARD SHALL BE OPAQUE (NON SEE THRU)

2. POLARIZATION KEY LENGTH.

3. MILLIMETERS ARE IN PARENTHESIS ( ).