PCI BIOS SPECIFICATION

Revision 2.1

August 26, 1994

REVISION	REVISION HISTORY	DATE
1.0	Original issue distributed by Intel	9/28/92
2.0	Updated to be in synch with PCI Bus Specification Rev 2.0	7/20/93
2.1	Added functions for PCI IRQ routing; Clarifications.	8/26/94

The PCI Special Interest Group disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does the PCI Special Interest Group make a commitment to update the information contained herein.

Contact the PCI Special Interest Group office to obtain the latest revision of the specification.

Questions regarding the PCI BIOS specification or membership in the PCI Special Interest Group may be forwarded to:

<div align="center">

PCI Special Interest Group
2575 NE Kathryn St #17
Hillsboro, OR   97124
(800) 433-5177 - Domestic
(503) 693-6232 - International
(503) 693-8344 - Fax
techsupp@pcisig.com

</div>

TABLE OF CONTENTS

1.   Introduction

1.1.   Purpose

This document describes the software interface presented by the PCI BIOS functions. This interface provides a hardware independent method of managing PCI devices in a host computer.

1.2.   Scope

This document is intended to provide enough information to software developers to utilize PCI devices 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 these BIOS functions for a particular system design.

1.3.   Related Documents

PCI Local Bus Specification, Revision 2.0 April 30, 1993

Standard BIOS 32-bit Service Directory Proposal, Revision 0.4 May 24, 1993 (available from Phoenix Technologies Ltd., Norwood, MA)

1.4.   Terms and Abbreviations

Bus Number	A number in the range 0 .. 255 that uniquely selects a PCI bus.
Configuration Space	A separate address space on PCI buses. Used for device identification and configuring devices into Memory and IO spaces.
Device ID	A predefined field in configuration space that (along with Vendor ID) uniquely identifies the device.
Device Number	A number in the range 0..31 that uniquely selects a device on a PCI bus.
Function Number	A number in the range 0..7 that uniquely selects a function within a multi-function PCI device.
Multi-function PCI device	A PCI device that contains multiple functions. For instance, a single device that provides both LAN and SCSI functions, and has a separate configuration space for each function is a multi-function device.
PCI	Acronym for Peripheral Component Interconnect bus.
Special Cycle	A specific PCI bus command used for broadcasting to all PCI devices on a bus.
Vendor ID	A predefined field in configuration space that (along with Device ID) uniquely identifies the device.

2. Functional Description

PCI BIOS functions provide a software interface to the hardware used to implement a PCI based system. Its primary usage is for generating operations in PCI specific address spaces (configuration space, and Special Cycles).

PCI BIOS functions are specified, and should be built, to operate in all modes of the X86 architecture. This includes real-mode, 16:16 protected mode (also known as 286 protect-mode), 16:32 protected mode (introduced with the 386), and 0:32 protected mode (also known as "flat" mode, wherein all segments start at linear address 0 and span the entire 4 Gbyte address space).

Access to the PCI BIOS functions for 16-bit callers is provided through Interrupt 1Ah. 32-bit (ie. protect mode) access is provided by calling through a 32-bit protect mode entry point. The PCI BIOS function code is B1h. Specific BIOS functions are invoked using a subfunction code. A user simply sets the host processors registers for the function and subfunction desired and calls the PCI BIOS software. Status is returned using the Carry flag ([CF]) and registers specific to the function invoked.

3. Assumptions and Constraints

3.1. ROM BIOS Location

The PCI BIOS functions are intended to be located within an IBM-PC compatible ROM BIOS.

3.2. Calling Conventions

The PCI BIOS functions use the X86 CPU's registers to pass arguments and return status. The caller must use the appropriate subfunction code.

These routines preserve all registers and flags except those used for return parameters. The CARRY FLAG [CF] will be altered as shown to indicate completion status. The calling routine will be returned to with the interrupt flag unmodified and interrupts will not be enabled during function execution. These routines are re-entrant. These routines require 1024 bytes of stack space and the stack segment must have the same size (ie. 16-bit or 32-bit) as the code segment.

The PCI BIOS provides a 16-bit real and protect mode interface and a 32-bit protect mode interface. The 16-bit interface is provided through PC/AT Int 1Ah software interrupt. The PCI BIOS Int 1Ah interface operates in either real mode, virtual-86 mode, or 16:16 protect mode. The BIOS functions may also be accessed through the industry standard entry point for INT 1Ah (physical address 000FFE6Eh) by simulating an INT instruction[1]. The INT 1Ah entry point supports 16-bit code only. Protect mode callers of this interface must set the CS selector base to 0F000h.

The protected mode interface supports 32-bit protect mode callers. The protected mode PCI BIOS interface is accessed by calling (not a simulated INT) through a protected mode entry point in the PCI BIOS. The entry point and information needed for building the segment descriptors are provided by the BIOS32 Service Directory (see section 3.3). 32-bit callers invoke the PCI BIOS routines using CALL FAR.

The PCI BIOS routines (for both 16-bit and 32-bit callers) must be invoked with appropriate privilege so that interrupts can be enabled/disabled and the routines can access IO space. Implementors of the PCI BIOS must assume that CS is execute-only and DS is read-only.

---

[1]Note that accessing the BIOS functions through the industry standard entry point will bypass any code that may have 'hooked' the INT 1Ah interrupt vector.

3.3. BIOS32 Service Directory[2]

Detecting the absence or presence of 32-bit BIOS services with 32-bit code can be problematic. Standard BIOS entry points cannot be called in 32-bit mode on all machines because the platform BIOS may not support 32-bit callers. This section describes a mechanism for detecting the presence of 32-bit BIOS services. While the mechanism supports the detection of the PCI BIOS, it is intended to be broader in scope to allow detection of any/all 32-bit BIOS services. The description of this mechanism, known as BIOS32 Service Directory, is provided in three parts; the first part specifies an algorithm for determining if the BIOS32 Service Directory exists on a platform, the second part specifies the calling interface to the BIOS32 Service Directory, and the third part describes how the BIOS32 Service Directory supports PCI BIOS detection.

3.3.1. Determining the existence of BIOS32 Service Directory

A BIOS which implements the BIOS32 Service Directory must embed a specific, contiguous 16-byte data structure, beginning on a 16-byte boundary somewhere in the physical address range 0E0000h - 0FFFFFh. A description of the fields in the data structure are given in Table 3.1.

Offset	Size	Description
0	4 bytes	Signature string in ASCII. The string is "_32_". This puts an 'underscore' at offset 0, a '3' at offset 1, a '2' at offset 2, and another 'underscore at' offset 3.
4	4 bytes	Entry point for the BIOS32 Service Directory. This is a 32-bit physical address.
8	1 byte	Revision level. This version has revision level 00h.
9	1 byte	Length. This field provides the length of this data structure in paragraph (i.e. 16-byte) units. This data structure is 16-bytes long so this field contains 01h.
0Ah	1 byte	Checksum. This field is a checksum of the complete data structure. The sum of all bytes must add up to 0.
0Bh	5 bytes	Reserved. Must be zero.

Table 3.1

Clients of the BIOS32 Service Directory should determine its existence by scanning 0E0000h to 0FFFF0h looking for the ASCII signature and a valid, checksummed data structure. If the data structure is found, the BIOS32 Service Directory can be accessed through the entry point provided in the data structure. If the data structure is not found, then the BIOS32 Service Directory (and also the PCI BIOS) is not supported by the platform.

---

[2]This section describes a mechanism for detecting 32-bit BIOS services. This mechanism is being proposed as an industry standard and is described by the document **Standard BIOS 32-bit Service Directory Proposal**, Revision 0.4 May 24, 1993 available from Phoenix Technologies Ltd., Norwood, MA.

3.3.2. Calling Interface for BIOS32 Service Directory

The BIOS32 Service Directory is accessed by doing a CALL FAR to the entry point provided in the Service data structure (see previous section). There are several requirements about the calling environment that must be met. The CS code segment selector and the DS data segment selector must be set up to encompass the physical page holding the entry point as well as the immediately following physical page. They must also have the same base. Platform BIOS writers must assume that CS is execute-only and DS is read-only. The SS stack segment selector must provide at least 1K of stack space. The calling environment must also allow access to IO space.

The BIOS32 Service Directory provides a single function to determine whether a particular 32-bit BIOS service is supported by the platform. All parameters to the function are passed in registers. Parameter descriptions are provided below. If a particular service is implemented in the platform BIOS, three values are returned. The first value is the base physical address of the BIOS service. The second value is the length of the BIOS service. These two values can be used to build the code segment selector and data segment selector for accessing the service. The third value provides the entry point to the BIOS service encoded as an offset from the base.

ENTRY:

[EAX]	Service Identifier. This is a four character string used to specifically identify which 32-bit BIOS Service is being sought.
[EBX]	The low order byte ([BL]) is the BIOS32 Service Directory function selector. Currently only one function is defined (with the encoding of zero) which returns the values provided below.
	The upper three bytes of [EBX] are reserved and must be zero on entry.

EXIT:

[AL]	Return code. 00h = Service corresponding to Service Identifier is present. 80h = Service corresponding to Service Identifier is not present 81h = Unimplemented function for BIOS Service Directory (i.e. BL has an unrecognized value).
[EBX]	Physical address of the base of the BIOS service.
[ECX]	Length of the BIOS service.
[EDX]	Entry point into BIOS service. This is an offset from the base provided in EBX.

3.4.   PCI BIOS 32-bit service

The BIOS32 Service Directory may be used to detect the presence of the PCI BIOS. The
Service Identifier for the PCI BIOS is "$PCI" (049435024h).

The 32-bit PCI BIOS functions must be accessed using CALL FAR. The CS and DS
descriptors must be setup to encompass the physical addresses specified by the Base and
Length parameters returned by the BIOS32 Service Directory. The CS and DS
descriptors must have the same base. The calling environment must allow access to IO
space and provide at least 1K of stack space. Platform BIOS writers must assume that CS
is execute-only and DS is read-only.

4.   Host Interface

4.1.   Identifying PCI Resources

The following group of functions allow the caller to determine first, if the PCI BIOS support is installed, and second, if specific PCI devices are present in the system.

4.1.1.   PCI BIOS Present

This function allows the caller to determine whether the PCI BIOS interface function set is present, and what the current interface version level is. It also provides information about what hardware mechanism for accessing configuration space is supported, and whether or not the hardware supports generation of PCI Special Cycles.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	PCI_BIOS_PRESENT

EXIT:

[EDX]	"PCI ", "P" in [DL], "C" in [DH], etc. There is a 'space' character in the upper byte.
[AH]	Present Status, 00h = BIOS Present IFF EDX set properly.
[AL]	Hardware mechanism
[BH]	Interface Level Major Version
[BL]	Interface Level Minor Version
[CL]	Number of last PCI bus in the system.
[CF]	Present Status, set = No BIOS Present, reset = BIOS Present IFF EDX set properly.

If the CARRY FLAG [CF] is cleared and AH is set to 00h, it is still necessary to examine the contents of [EDX] for the presence of the string "PCI" + (trailing space) to fully validate the presence of the PCI function set. [BX] will further indicate the version level, with enough granularity to allow for incremental changes in the code that don't affect the function interface. Version numbers are stored as Binary Coded Decimal (BCD) values. For example, Version 2.10 would be returned as a 02h in the [BH] registers and 10h in the [BL] registers. BIOS releases of the interface following this version of the specification will be Version 2.10 (BX = 0210h).

The value returned in [AL] identifies what specific HW characteristics the platform supports in relation to accessing configuration space and generating PCI Special Cycles (see Figure 1). The PCI Specification defines two HW mechanisms for accessing configuration space. Bits 0 and 1 of the value returned in [AL] specify which mechanism is supported by this platform. Bit 0 will be set (1) if Mechanism #1 is supported, and reset (0) otherwise. Bit 1 will be set (1) if Mechanism #2 is supported, and reset (0) otherwise. Bits 2,3,6 and 7 are reserved and returned as zeros.

The PCI Specification also defines HW mechanisms for generating Special Cycles. Bits 4 and 5 of the value return in [AL] specify which mechanism is supported (if any). Bit 4 will be set (1) if the platform supports Special Cycle generation based on Config Mechanism #1, and reset (0) otherwise. Bit 5 will be set (1) if the platform supports Special Cycle generation based on Config Mechanism #2, and reset (0) otherwise.
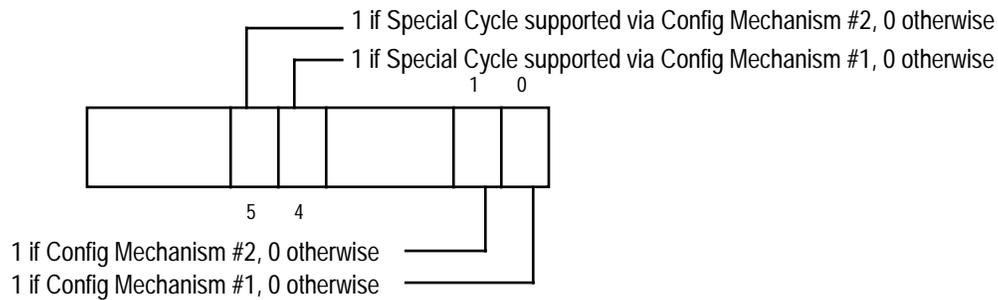
Figure 1. Layout of value returned in [AL]

The value returned in [CL] specifies the number of the last PCI bus in the system. PCI buses are numbered starting at zero and running up to the value specified in CL.

4.1.2. Find PCI Device

This function returns the location of PCI devices that have a specific Device ID and Vendor ID. Given a Vendor ID, Device ID and an Index (N), the function returns the Bus Number, Device Number, and Function Number of the Nth Device/Function whose Vendor ID and Device ID match the input parameters.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	FIND_PCI_DEVICE
[CX]	Device ID (0...65535)
[DX]	Vendor ID (0...65534)
[SI]	Index (0...N)

EXIT:

[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits, Function Number in bottom 3 bits
[AH]	Return Code: SUCCESSFUL DEVICE_NOT_FOUND BAD_VENDOR_ID
[CF]	Completion Status, set = error, cleared = success.

Calling software can find all devices having the same Vendor ID and Device ID by making successive calls to this function starting with Index set to zero, and incrementing it until the return code is 'DEVICE_NOT_FOUND'. A return code of BAD_VENDOR_ID indicates that the passed in Vendor ID value (in [DX]) had an illegal value of all 1's.

Values returned by this function upon successful completion must be the actual values used to access the PCI device if the INT 1Ah routines are by-passed in favor of the direct I/O mechanisms described in the PCI Specification.

4.1.3. Find PCI Class Code

This function returns the location of PCI devices that have a specific Class Code. Given a Class Code and a Index (N), the function returns the Bus Number, Device Number, and Function Number of the Nth Device/Function whose Class Code matches the input parameters.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	FIND_PCI_CLASS_CODE
[ECX]	Class Code (in lower three bytes)
[SI]	Index (0...N)

EXIT:

[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in bottom 3 bits
[AH]	Return Code:
	SUCCESSFUL
	DEVICE_NOT_FOUND
[CF]	Completion Status, set = error, cleared = success.

Calling software can find all devices having the same Class Code by making successive calls to this function starting with Index set to zero, and incrementing it until the return code is 'DEVICE_NOT_FOUND'.

4.2.   PCI Support Functions

The following functions provide support for several PCI specific operations.

4.2.1.   Generate Special Cycle

This function allows for generation of PCI special cycles. The generated special cycle will be broadcast on a specific PCI bus in the system.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	GENERATE_SPECIAL_CYCLE
[BH]	Bus Number (0...255)
[EDX]	Special Cycle Data

EXIT:

[AH]	Return Code:
	SUCCESSFUL
	FUNC_NOT_SUPPORTED
[CF]	Completion Status, set = error, reset = success

4.2.2. Get PCI Interrupt Routing Options

Description:

Logical input parameters:

    RouteBuffer   Pointer to buffer data structure

This routine returns the PCI interrupt routing options available on the system motherboard and also the current state of what interrupts are currently exclusively assigned to PCI. Routing information is returned in a data buffer that contains an IRQ Routing for each PCI device or slot. The format of an entry in the IRQ routing table is shown in Table 4-1.

Two values are provided for each PCI interrupt pin in every slot. One of these values is a bit-map that shows which of the standard AT IRQs this PCI interrupt can be routed to. This provides the routing options for one particular PCI interrupt pin. In this bit-map, bit 0 corresponds to IRQ0, bit 1 to IRQ1, etc. A '1' bit in this bit-map indicates a routing is possible; a '0' bit indicates no routing is possible. The second value is a 'link' value that provides a way of specifying which PCI interrupt pins are wire-OR'ed together on the motherboard. Interrupt pins that are wired together must have the same 'link' value in their table entries. Values for the 'link' field are arbitrary except that the value zero

Offset	Size	Description
0	byte	PCI Bus number
1	byte	PCI Device number (in upper 5 bits)
2	byte	Link value for INTA#
3	word	IRQ bit-map for INTA#
5	byte	Link value for INTB#
6	word	IRQ bit-map for INTB#
8	byte	Link value for INTC#
9	word	IRQ bit-map for INTC#
11	byte	Link value for INTD#
12	word	IRQ bit-map for INTD#
14	byte	Slot Number
15	byte	Reserved

Table 4-1 Layout of IRQ routing table entry.

indicates that the PCI interrupt pin has no connection to the interrupt controller[3].

The Slot Number value at the end of the structure is used to communicate whether the table entry is for a motherboard device or an add-in slot. For motherboard devices, Slot Number should be set to zero. For add-in slots, Slot Number should be set to a value that corresponds with the physical placement of the slot on the motherboard. This provides a way to correlate physical slots with PCI Device numbers. Values (with the exception of

---

[3]This is typically used for motherboard devices who have only an IRQA# line and not IRQB#, IRQC# or IRQD#.

00h) are OEM specific.[4] For end user ease-of-use, slots in the system should be clearly labeled (e.g. solder mask, back-panel, etc.).

This routine requires one parameter, *RouteBuffer,* that is a far pointer to the data structure shown below.

```
typedef struct
    {
        WORD        BufferSize;
        BYTE   FAR * DataBuffer;
    } IRQRoutingOptionsBuffer;
```

where

> *BufferSize*:   A word size value providing the size of the data buffer. If the buffer is too small, the routine will return with status of BUFFER_TOO_SMALL, and this field will be updated with the required size. To indicate that the running PCI system does not have *any* PCI devices, this function will update the *BufferSize* field to zero. On successful completion this field is updated with the size (in bytes) of the data returned.

> *DataBuffer*: Far pointer to the buffer containing PCI interrupt routing information for all motherboard devices and slots.

This routine also returns information about which IRQs are currently dedicated for PCI usage. This information is returned as a bit map where a set bit indicates that the IRQ is dedicated to PCI and not available for use by devices on other buses. Note that if an IRQ is routed such that it can be used by PCI devices and other devices the corresponding bit in the bit map should not be set. The function returns this information in the [BX] register where bit 0 corresponds to IRQ0, bit 1 - IRQ1, etc. The caller must initialize [BX] to zero before calling this routine.

---

[4]For example, a system with 4 ISA slots and 3 PCI slots arranged as 3-ISA, 3-PCI, 1-ISA may choose to start numbering the slots at the 3-ISA end in which case the PCI slot numbers would be 4, 5 and 6. If slot numbering started at the 1-ISA end, PCI slot numbers would be 2, 3, and 4.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	GET_IRQ_ROUTING_OPTIONS
[BX]	Must be initialized to 0000h.
[DS]	Segment or Selector for BIOS data.
	For 16-bit code the real-mode segment or PM selector must resolve to physical address 0F0000h and have a limit of 64K.
	For 32-bit code see section 3.4.
[ES]	Segment or Selector for *RouteBuffer* parameter
[DI] for 16-bit code	Offset for *RouteBuffer* parameter.
[EDI] for 32-bit code	

EXIT:

[AH]	Return Code:
	SUCCESSFUL
	BUFFER_TOO_SMALL
	FUNC_NOT_SUPPORTED
[BX]	IRQ bitmap indicating which IRQs are exclusively dedicated to PCI devices.
[CF]	Completion Status, set = error, cleared = success

4.2.3. Set PCI Hardware Interrupt

Description:

This function is intended to be used by a system-wide configuration utility or a PNP OS. This function should never be called by device drivers or expansion ROM code.

Logical input parameters:

BusDev	Bus number and device number
IntPin	PCI Interrupt Pin (INTA .. INTD)
IRQNum	IRQ Number (0-15)

This routine causes the specified hardware interrupt (IRQ) to be connected to the specified interrupt pin of a PCI device. It makes the following assumptions:

1) The caller is responsible for all error checking to ensure no resource conflict exits between the specific hardware interrupt assigned to the PCI device and any other hardware interrupt resource in the system.

2) The caller is responsible for ensuring that the specified interrupt is configured properly (level triggered) in the interrupt controller. If the system contains hardware outside of the interrupt controller that controls interrupt triggering (edge/level) then the callee (i.e.; the BIOS) is responsible for setting that hardware to level triggered for the specified interrupt.

3) The caller is responsible for updating PCI configuration space (ie. Interrupt Line registers) for all effected devices.

4) The caller must be aware that changing IRQ routing for one device will also change the IRQ routing for other devices whose INTx# pins are WIRE-ORed together (ie. they have the same *link* field in the Get Routing Options call).

If the requested interrupt cannot be assigned to the specified PCI device then **SET_FAILED** status is returned. This routine immediately effects the interrupt routing and does nothing to remember the routing for the next system boot.

The *BusDev* parameter specifies the PCI bus and device numbers for the PCI device/slot being modified. The high-order byte of *BusDev* contains the PCI bus number. The device number is provided in the top five bits of the low-order byte of *BusDev*. For example, to specify device 6 on PCI bus 2 the *BusDev* parameter would be 0x0230.

The *IntPin* parameter specifies which interrupt pin (INTA#,..,INTD#) of the specified PCI device/slot is effected by this call. A value of 0x0A corresponds to INTA#, 0x0B to INTB#, etc.

The *IRQNum* parameter specifies which IRQ input is to be connected to the PCI interrupt pin. This parameter can have values of 0..15 specifying IRQ0 thru IRQ15 respectively.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	SET_PCI_HW_INT
[CL]	*IntPin* parameter. Valid values 0Ah..0Dh
[CH]	*IRQNum* parameter. Valid values 0..0Fh
[BX]	*BusDev* parameter. [BH] holds bus number, [BL] holds Device (upper five bits) and Function (lower 3 bits) numbers.
[DS]	Segment or Selector for BIOS data. For 16-bit code the real-mode segment or PM selector must resolve to physical address 0F0000h and have a limit of 64K. For 32-bit code see section 3.4.

EXIT:

[AH]	Return Code: SUCCESSFUL SET_FAILED FUNC_NOT_SUPPORTED
[CF]	Completion Status, set = error, cleared = success

4.3.   Accessing Configuration Space

4.3.1. Read Configuration Byte

This function allows reading individual bytes from the configuration space of a specific device.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	READ_CONFIG_BYTE
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits, Function Number in lower 3 bits
[DI]	Register Number (0...255)

EXIT:

[CL]	Byte Read
[AH]	Return Code: SUCCESSFUL
[CF]	Completion Status, set = error, reset = success

4.3.2. Read Configuration Word

This function allows reading individual words from the configuration space of a specific device. The Register Number parameter must be a multiple of two (i.e., bit 0 must be set to 0).

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	READ_CONFIG_WORD
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in lower 3 bits
[DI]	Register Number (0,2,4,...254)

EXIT:

[CX]	Word Read
[AH]	Return Code:
	SUCCESSFUL
	BAD_REGISTER_NUMBER
[CF]	Completion Status, set = error, reset = success

4.3.3. Read Configuration Dword

This function allows reading individual dwords from the configuration space of a specific device. The Register Number parameter must be a multiple of four (i.e., bits 0 and 1 must be set to 0).

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	READ_CONFIG_DWORD
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in lower 3 bits
[DI]	Register Number (0,4,8,...252)

EXIT:

[ECX]	Dword read.
[AH]	Return Code:
	SUCCESSFUL
	BAD_REGISTER_NUMBER
[CF]	Completion Status, set = error, reset = success

4.3.4. Write Configuration Byte

This function allows writing individual bytes to the configuration space of a specific device.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	WRITE_CONFIG_BYTE
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in lower 3 bits
[DI]	Register Number (0...255)
[CL]	Byte Value to Write

EXIT:

[AH]	Return Code:
	SUCCESSFUL
[CF]	Completion Status, set = error, reset = success

4.3.5. Write Configuration Word

This function allows writing individual words from the configuration space of a specific device. The Register Number parameter must be a multiple of two (i.e., bit 0 must be set to 0).

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	WRITE_CONFIG_WORD
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in lower 3 bits
[DI]	Register Number (0,2,4,...254)
[CX]	Word Value to Write

EXIT:

[AH]	Return Code:
	SUCCESSFUL
	BAD_REGISTER_NUMBER
[CF]	Completion Status, set = error, reset = success

4.3.6. Write Configuration Dword

This function allows writing individual dwords from the configuration space of a specific device. The Register Number parameter must be a multiple of four (i.e., bits 0 and 1 must be set to 0).

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	WRITE_CONFIG_DWORD
[BH]	Bus Number (0...255)
[BL]	Device Number in upper 5 bits,
	Function Number in lower 3 bits
[DI]	Register Number (0,4,8,...252)
[ECX]	Dword value to write.

EXIT:

[AH]	Return Code:
	SUCCESSFUL
	BAD_REGISTER_NUMBER
[CF]	Completion Status, set = error, reset = success

APPENDIX A: Function List

FUNCTION	AH	AL
PCI_FUNCTION_ID	B1h	
PCI_BIOS_PRESENT		01h
FIND_PCI_DEVICE		02h
FIND_PCI_CLASS_CODE		03h
GENERATE_SPECIAL_CYCLE		06h
READ_CONFIG_BYTE		08h
READ_CONFIG_WORD		09h
READ_CONFIG_DWORD		0Ah
WRITE_CONFIG_BYTE		0Bh
WRITE_CONFIG_WORD		0Ch
WRITE_CONFIG_DWORD		0Dh
GET_IRQ_ROUTING_OPTIONS		0Eh
SET_PCI_IRQ		0Fh

APPENDIX B: Return Code List

RETURN CODES	AH
SUCCESSFUL	00h
FUNC_NOT_SUPPORTED	81h
BAD_VENDOR_ID	83h
DEVICE_NOT_FOUND	86h
BAD_REGISTER_NUMBER	87h
SET_FAILED	88h
BUFFER_TOO_SMALL	89h